



Universidad Internacional de La Rioja
Escuela Superior de Ingeniería y Tecnología

Grado en Ingeniería Informática

Sistema Blockchain de Almacenamiento Distribuido para Dispositivos IoT

Trabajo fin de estudio presentado por:	José Ignacio Bravo Vicente
Director/a:	José Arturo Mora Soto
Fecha:	16/07/2025
Repositorio del código fuente:	https://github.com/joseigbv/dfs3

Resumen

Se presenta el diseño e implementación de un sistema de almacenamiento distribuido como alternativa para la gestión de archivos en entornos IoT. La solución se alinea con los principios de la Web 3.0, promoviendo descentralización, soberanía del usuario y transparencia. Se apoya en dispositivos de bajo consumo, utilizados como nodos colaborativos capaces de almacenar, compartir y recuperar archivos sin depender de infraestructuras centralizadas.

El sistema implementa una arquitectura distribuida abierta. Usuarios, nodos y archivos se identifican mediante hashes criptográficos únicos. Los datos se almacenan cifrados y de forma redundante en varios nodos, lo que mejora la disponibilidad y el acceso en redes P2P. Los metadatos y operaciones se registran de forma inmutable en la Tangle de IOTA, que actúa como capa de control descentralizada. Para adaptarse a dispositivos con recursos limitados, se priorizan mecanismos de sincronización ligeros, como la publicación de eventos en IOTA y su propagación mediante MQTT. De este modo, se evita recurrir a algoritmos de consenso complejos que penalicen el rendimiento. La interacción se realiza mediante API REST y una interfaz web sencilla.

El modelo demuestra ser viable y se ajusta a los principios de seguridad, resiliencia y soberanía del dato. Se plantea como una solución libre y sostenible, aplicable en entornos domésticos, industriales o de investigación. Promueve la participación de pequeños nodos distribuidos globalmente, con un impacto positivo en eficiencia energética y acceso equitativo a la tecnología. Como línea futura, se contempla su evolución hacia modelos de computación en la niebla, donde los nodos, también ejecuten tareas distribuidas de procesamiento. Para incentivar esta colaboración, se sugiere la integración de un modelo económico basado en tokens que recompense la participación.

Palabras clave: almacenamiento distribuido, IoT, Blockchain, Web 3.0, P2P

Abstract

This work presents the design and implementation of a distributed storage system as an alternative for file management in IoT environments. The solution aligns with the principles of Web 3.0, promoting decentralization, user sovereignty and transparency. It relies on low-power devices acting as collaborative nodes capable of storing, sharing and retrieving files without depending on centralized infrastructures.

The system implements an open distributed architecture. Users, nodes and files are identified using unique cryptographic hashes. Data is encrypted and redundantly stored across multiple nodes, improving availability and access in P2P networks. Metadata, including operations, is immutably recorded on the IOTA Tangle, which serves as a decentralized control layer. To suit resource-constrained devices, lightweight synchronization mechanisms are prioritized, such as publishing events to IOTA and propagating them via MQTT. This avoids the need for complex consensus algorithms that would degrade performance. Interaction with the system is enabled through a REST API and a simple web interface.

The proposed model proves to be viable and adheres to principles of security, resilience, and data sovereignty. It is presented as a free and sustainable solution, suitable for domestic, industrial or research contexts. It promotes the participation of small nodes distributed globally, with a positive impact on energy efficiency and equitable access to technology. As a future direction, the system could evolve towards fog computing models, where nodes also perform distributed processing tasks. To incentivize this collaboration, the integration of a token-based economic model is proposed, enabling nodes to be rewarded for their active participation.

Keywords: Distributed Storage, IoT, Blockchain, Web 3.0, P2P

Índice de contenidos

1.	Introducción.....	1
1.1.	Motivación	2
1.2.	Planteamiento del trabajo.....	3
1.3.	Estructura del trabajo.....	4
2.	Contexto y estado del arte.....	5
2.1.	Análisis del contexto	5
2.2.	Estado del arte	6
2.2.1.	Introducción a la Web 3.0	6
2.2.2.	Redes Peer-to-Peer (P2P)	7
2.2.3.	Introducción a blockchain	7
2.2.4.	Introducción al almacenamiento distribuido	12
2.2.5.	Introducción a IoT.....	16
2.2.6.	IoT y blockchain	20
2.2.7.	Enfoques de almacenamiento distribuido para IoT	21
2.2.8.	Tecnologías blockchain ligeras para IoT	23
2.2.9.	Ejemplos de casos de uso	27
2.2.10.	Desafíos	33
2.2.11.	Conclusión de estado del arte	33
3.	Objetivos y metodología de trabajo	35
3.1.	Objetivo general	35
3.2.	Objetivos específicos	35
3.3.	Metodología de trabajo.....	36
4.	Ánalisis de requisitos	38
4.1.	Descripción general.....	38

4.2.	Requisitos funcionales.....	42
4.3.	Requisitos no funcionales.....	43
4.4.	Restricciones y limitaciones	43
5.	Diseño del sistema	46
5.1.	Arquitectura general	46
5.1.1.	Comunicación entre componentes	47
5.2.	Justificación de elección tecnológica.....	48
5.2.1.	IOTA Tangle	48
5.2.2.	MQTT.....	49
5.2.3.	SQLite	50
5.2.4.	Cliente Web	50
5.3.	Modelo de casos de uso	51
5.4.	Modelo de clases.....	52
5.5.	Modelo de datos	54
5.6.	Consideraciones de seguridad.....	55
5.7.	Consideraciones de sincronización y replicación	56
6.	Implementación	59
6.1.	Entorno de desarrollo.....	59
6.1.1.	Tecnologías y herramientas.....	59
6.1.2.	Infraestructura	62
6.2.	Desarrollo e implementación del servicio	71
6.2.1.	Estructura del proyecto	71
6.2.2.	Implementación de API REST	73
6.2.3.	Gestión de almacenamiento	77
6.2.4.	Gestión de eventos distribuidos	79

6.2.5. Uso de cachés y optimizaciones en backend.....	81
6.3. Implementación del cliente web.....	82
6.3.1. Estructura de archivos.....	82
6.3.2. Generación y gestión de claves en el navegador	83
6.3.3. Registro de usuarios	83
6.3.4. Autenticación mediante desafío y firma digital.....	84
6.3.5. Subida de ficheros	85
6.3.6. Descarga y descifrado.....	86
6.3.7. Compartición de ficheros	88
7. Pruebas y validación	91
7.1. Pruebas unitarias.....	91
7.2. Pruebas funcionales	92
7.3. Pruebas de integración.....	93
7.4. Pruebas de sistema	94
7.5. Pruebas de seguridad	98
7.6. Resumen de validación de requisitos	99
8. Conclusiones y trabajo futuro.....	100
8.1. Conclusiones del trabajo	100
8.2. Líneas de trabajo futuro	101
Referencias bibliográficas	103
Anexo A. Mecanismos de consenso blockchain	108
Anexo B. Tipos de redes blockchain	110
Anexo C. Principales Redes blockchain.....	111
Anexo D. Desafíos del almacenamiento distribuido blockchain.....	113
Anexo E. Casos de uso extendidos	115

Código UML	115
CU01: Alta de usuario	116
CU02: Autenticación de usuario	118
CU03: Alta de nodo.....	120
CU04: Actualización de estado de nodo	122
CU05: Subida de fichero	124
CU06: Descarga de fichero	127
CU07: Compartición de fichero	128
CU08: Listado de ficheros	130
CU09: Renombrado de fichero virtual.....	132
CU10: Borrado de fichero	133
CU11: Replicación de fichero.....	135
Anexo F. Modelo de clases.....	137
Código UML	137
Clases.....	138
Usuario.....	138
Nodo	138
Fichero	139
Entrada.....	139
Evento	139
Anexo G. Diagrama de actividades	140
Subida de fichero.....	140
Descarga de fichero	141
Compartición de fichero	143
Replicación de fichero	145

Alta de usuario.....	146
Alta de nodo	148
Anexo H. Modelo de base de datos	150
Relaciones UML	150
Tablas.....	151
Código SQL (SQLite)	153
Anexo I. Lógica de replicación de ficheros	155
Algoritmo	155
Consideraciones.....	155
Criterios y parámetros de replicación	156
Filtrado de nodos candidatos.....	156
Ordenación determinista de candidatos.....	156
Selección de nodos replicadores.....	157
Lógica de auto inclusión.....	158
Anexo J. Límite de tamaño de fichero	159
Anexo K. API REST.....	160
Autenticación y registro	160
POST /auth/register	160
POST /auth/challenge	160
POST /auth/verify	161
Gestión de usuarios	161
GET /users	161
GET /users/{user_id}.....	162
Nodos	162
GET /nodes.....	162

GET /nodes/{node_id}	163
Ficheros	163
GET /files.....	163
POST /files.....	163
GET /files/{filename}.....	165
GET /files/{file_id}/meta	165
GET /files/{file_id}/data	166
PATCH /files/{filename}	166
DELETE /files/{filename}	166
POST /files/share	166
Eventos	167
GET /events.....	167
GET /events/{node_id}.....	167
Anexo L. Metainformación de fichero	168
Anexo M. Definición de eventos.....	170
Usuarios.....	171
Nodos	171
Ficheros	172
Anexo N. Formato de mensaje MQTT.....	175
Anexo O. Fichero de configuración	176
Anexo P. Erasure Coding	178
Principio de operación.....	178
Esquema propuesto.....	179
Ventajas de la propuesta	179
Consideraciones	180

Detalle de implementación	180
Estructura de datos.....	180
Flujo de replicación	181
Actualización de estado de replicación.....	182
Limitaciones y consideraciones adicionales.....	182
Estrategia inicial de asignación de fragmentos por índice fijo.....	183
Anexo Q. Estrategia de replicación con incentivos económicos	184
Selección determinista de nodos replicadores.....	184
Verificación de almacenamiento	185
Integración con contratos inteligentes de IOTA (ISCP).....	185
Anexo R. Autenticación dual mediante JWT.....	187
Objetivo	187
Principios del modelo propuesto.....	187
Flujo de autenticación	188
Estructura del JWT.....	189
Ventajas	189
Conclusión	190
Consideraciones adicionales.....	190
Anexo S. VPS de desarrollo y pruebas.....	191
Listado de servicios.....	191
Características Hardware.....	192
Configuración Nginx	193
Anexo T. Coste estimado del proyecto.....	195
Anexo U. Impacto social, económico y medioambiental.....	196
Impacto social.....	196

Impacto económico	196
Impacto medioambiental	197
Conclusión	197
Índice de acrónimos	198

Índice de figuras

Figura 1. Evolución de Internet hacia un modelo descentralizado	1
Figura 2. Un fallo en Azure provoca el colapso de aeropuertos, hospitales y policía	5
Figura 3. Principales características de la Web 3.0	6
Figura 4. Estructura de bloques característica de una blockchain	8
Figura 5. Arquitectura de IPFS frente a un modelo Cliente-Servidor tradicional	13
Figura 6. IoT World Forum Reference Model.....	17
Figura 7. Arquitectura de tres capas en fog Computing	20
Figura 8. Generación de bloques, validación y adición en blockchain.....	21
Figura 9: Post en X del creador de la primera web hospedada en Cardano	27
Figura 10. Estructura de datos Merkle tree	29
Figura 11. Arquitectura blockchain de Modum.io AG.....	30
Figura 12. Sistema de compartición de ficheros dfs3	38
Figura 13. Identidad digital dfs3 basada en pares de claves.....	39
Figura 14. Acceso al sistema de ficheros de dfs3	40
Figura 15. Sistema de replicación por eventos y publicación en IOTA dfs3	41
Figura 16. Propuesta de escalabilidad horizontal y resiliencia en dfs3.....	41
Figura 17. Arquitectura general de la solución	46
Figura 18. Diagrama de la arquitectura de la solución.....	48
Figura 19. Diagrama general de casos de uso	51
Figura 20. Diagrama de clases UML	53
Figura 21. Diagrama entidad relación del modelo de datos	55
Figura 22. Estrategia de replicación en dfs3	58
Figura 23. Cluster de tarjetas Orange Pi One usado para las pruebas.....	63
Figura 24. Primer acceso a Armbian	64

Figura 25. Descarga de S.O. de los nodos IoT	69
Figura 26. Pantalla principal Armbian	70
Figura 27. Estructura de carpetas dfs3 en GitHub	72
Figura 28. Interfaz OpenAPI autogenerada.....	73
Figura 29. Ejemplo de estructura de ficheros dfs3	78
Figura 30. Ejemplo de entrada virtual en dfs3	78
Figura 31. Ejemplo de entrada virtual con metadatos.....	78
Figura 32. Evento publicado en la red IOTA.....	80
Figura 33. Cliente web: listado de ficheros	82
Figura 34. Cliente web: registro de usuarios.....	84
Figura 35. Cliente web: autenticación de usuario.....	85
Figura 36. Cliente web: subida de ficheros	86
Figura 37: Cliente web: descarga de fichero	88
Figura 38. Cliente web: compartición de fichero	89
Figura 39. Evolución de uso de CPU y memoria durante 20 peticiones concurrentes	97
Figura 40. Carga estimada de CPU y memoria por número de nodos activos	97
Figura 41. Subida de fichero	141
Figura 42. Descarga de fichero.....	142
Figura 43. Compartición de fichero.....	144
Figura 44. Replicación de fichero	145
Figura 45. Alta de usuario	147
Figura 46. Alta de nodo	148
Figura 47. Selección de nodo backend en dfs3	192

Índice de tablas

Tabla 1. Principales mecanismos de consenso en blockchain	10
Tabla 2. Tipos de redes en blockchain	11
Tabla 3. Redes blockchain más extendidas	12
Tabla 4. Almacenamiento distribuido frente a centralizado o Cloud.....	12
Tabla 5. Resumen de los principales desafíos a los que se enfrenta el proyecto	33
Tabla 6. Comparativa de las soluciones de almacenamiento analizadas.....	34
Tabla 7. Planificación temporal del trabajo	37
Tabla 8. Listado de requisitos funcionales definidos para dfs3	42
Tabla 9. Listado de requisitos no funcionales	43
Tabla 10. Principales limitaciones y restricciones del proyecto.....	44
Tabla 11. Listado de casos de uso	51
Tabla 12. Entidades del modelado de clases	53
Tabla 13. Entidades del modelo de datos	54
Tabla 14. Listado de certificados TLS necesarios	68
Tabla 15. Listado de los endpoints en la API REST	74
Tabla 16. Estructura de los principales módulos del backend	75
Tabla 17. Estructura modular del sistema de gestión de eventos	79
Tabla 18. Estructura general de un evento	80
Tabla 19. Contenido de la carpeta ‘webclient’.....	82
Tabla 20. Listado de pruebas unitarias	92
Tabla 21. Listado de pruebas funcionales	92
Tabla 22. Listado de pruebas de integración	93
Tabla 23. Listado de pruebas de rendimiento	94
Tabla 24. Comportamiento del sistema ante cargas concurrentes	96

Tabla 25. Listado de pruebas de seguridad.....	98
Tabla 26. CU01 - Alta de usuario.....	116
Tabla 27. CU02 - Autenticación de usuario.....	118
Tabla 28. CU03 - Alta de nodo	120
Tabla 29. CU04 - Actualización de estado de nodo.....	122
Tabla 30. CU05 - Subida de fichero	124
Tabla 31. CU06 - Descarga de fichero	127
Tabla 32. CU07 - Compartición de fichero	128
Tabla 33. CU08 - Listado de ficheros.....	130
Tabla 34. CU09 - Renombrado de entrada.....	132
Tabla 35. CU10 - Borrado de fichero	133
Tabla 36. CU11 - Replicación de fichero	135
Tabla 37. Tabla 'users'.....	151
Tabla 38. Tabla 'nodes'	151
Tabla 39. Tabla 'files'	152
Tabla 40. Tabla 'entries'	152
Tabla 41. Tabla 'events'	153
Tabla 42. Campos de POST /files.....	164
Tabla 43. Metainformación de fichero	168
Tabla 44. Estructura general de un evento	170
Tabla 45. Eventos relativos a usuario.....	171
Tabla 46. Eventos relativos a nodo	172
Tabla 47. Eventos relativos a fichero	172
Tabla 48. Estructura del mensaje MQTT usado para la sincronización entre nodos	175
Tabla 49. Descripción de los campos de node.json	176

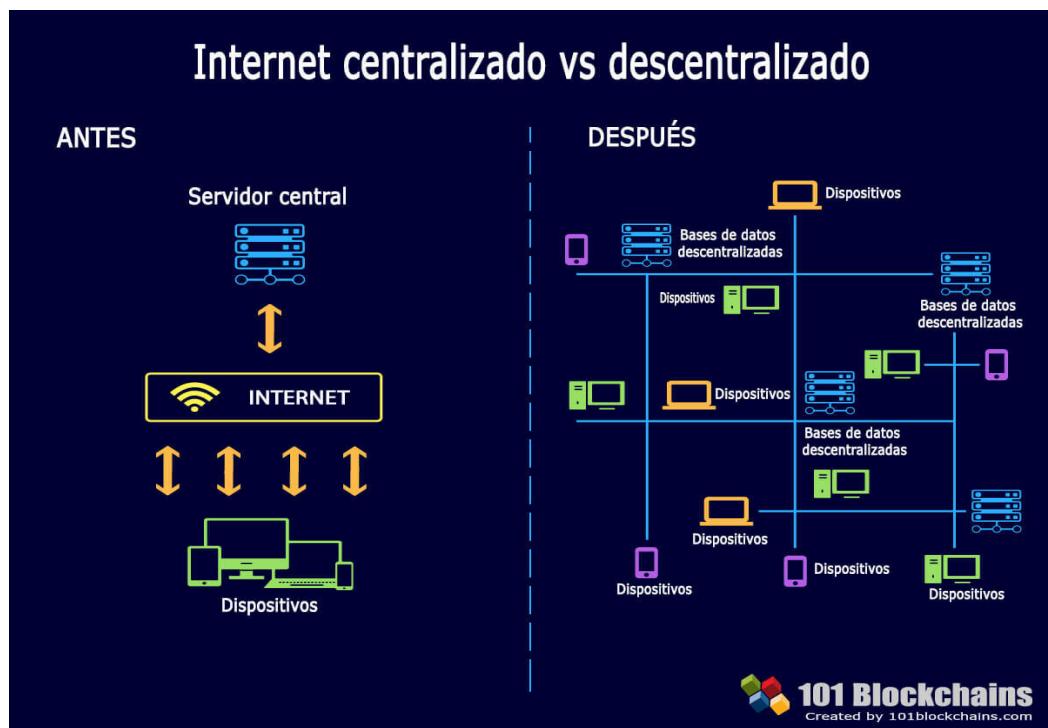
Tabla 50. Descripción del objeto keys en node.json	176
Tabla 51. Funciones del contrato inteligente propuestas para incentivo económico	185
Tabla 53. Características HW del VPS	192
Tabla 54. Desglose del coste estimado del proyecto	195

1. Introducción

Actualmente la tecnología se encuentra en una etapa de profunda transformación. Esta evolución, impulsada por la convergencia de tendencias como Web 3.0, IoT (*Internet of Things*), arquitecturas distribuidas y tecnologías *blockchain*, cambia nuestra forma de interactuar con los sistemas de información, especialmente en cuanto a almacenamiento, procesamiento y control de los datos.

La **Web 3.0**, en contraposición a los modelos tradicionales dominados por grandes proveedores de servicios como AWS, Google Cloud o Azure, **propone una internet más descentralizada, semántica y centrada en el usuario**. El nuevo enfoque busca devolver a los usuarios el control, la propiedad y la localización de sus datos, permitiéndoles decidir cómo, cuándo y con quién compartir su información, sin renunciar a principios fundamentales como la confidencialidad, integridad y disponibilidad (Krause, 2023). La **Figura 1** compara el modelo de Internet centralizado tradicional, basado en servidores únicos, con el enfoque descentralizado característico de la Web 3.0, donde los datos se distribuyen.

Figura 1. Evolución de Internet hacia un modelo descentralizado



Fuente: Rodriguez (2018).

IoT ha llevado las capacidades computacionales a dispositivos cada vez más pequeños, asequibles y energéticamente eficientes. Algunos como *Raspberry Pi*, ya no están limitados al ámbito doméstico o educativo: se integran activamente en entornos industriales, logísticos, o sanitarios, facilitando la monitorización en tiempo real. Su capacidad para actuar como nodos dentro de redes distribuidas abre la puerta a nuevos e interesantes modelos de computación colaborativa (Salih et al., 2022).

Las arquitecturas P2P (*Peer-to-peer*) resurgen con fuerza como alternativa a los modelos cliente-servidor. Proporcionan resiliencia frente a fallos, mayor eficiencia mediante paralelización y un notable ahorro de costes al prescindir de infraestructuras centralizadas. Distribuyendo los datos entre nodos, se mejora la disponibilidad, se reducen los cuellos de botella y se mitiga el riesgo de un único punto de fallo (Spiceworks, 2023).

Blockchain se consolida como una capa de control descentralizada capaz de alcanzar consenso sin necesidad de terceros. Gracias a su naturaleza inmutable y auditible, es especialmente útil para registrar transacciones, cambios en los datos, permisos y eventos relevantes. El uso de algoritmos de consenso ligeros como PoA (*Proof of Authority*) o PoS (*Proof of Service*), permite su integración incluso en redes IoT con recursos limitados (Khalil et al., 2020).

El proyecto explora la combinación de estas tecnologías para crear un sistema de almacenamiento distribuido. Se integran dispositivos IoT, arquitecturas P2P y tecnología blockchain con el objetivo de ofrecer una solución segura, resiliente, económica y auditable.

1.1. Motivación

La dependencia de infraestructuras centralizadas para almacenar y gestionar datos plantea desafíos clave en términos de privacidad, trazabilidad, disponibilidad y, sobre todo, soberanía digital. Esta problemática es especialmente crítica en sectores donde la integridad y la auditoría son requisitos legales.

El uso de soluciones centralizadas puede generar costes elevados y dependencia de terceros. En sectores regulados como el farmacéutico, la gestión de datos debe garantizar no solo la confidencialidad, integridad y disponibilidad, sino también la trazabilidad completa. Marcos

normativos como GxP¹ (*Good x Practice*) o la regulación 21 CFR Part 11² exigen sistemas capaces de registrar quién hizo qué, cuándo, cómo y por qué.

Los modelos tradicionales de almacenamiento centralizado presentan limitaciones en lo que respecta al control y la propiedad del dato, la resistencia a fallos, la transparencia operativa y la dependencia de terceros. La necesidad de soluciones alternativas invita a explorar nuevos enfoques.

Este proyecto surge del potencial que ofrece la integración de tecnologías emergentes como IoT, P2P y blockchain. Propone un sistema de almacenamiento distribuido de bajo coste que sea eficiente y seguro. Puede aplicarse tanto en contextos industriales fuertemente regulados como en entornos domésticos o de investigación.

1.2. Planteamiento del trabajo

Este trabajo responde a la necesidad de explorar nuevas formas de almacenamiento que respondan a los desafíos de seguridad actuales. En lugar de depender de infraestructuras centralizadas, se propone un sistema de almacenamiento distribuido, concebido para ejecutarse sobre dispositivos IoT de bajo consumo y basado en tecnologías propias de la Web 3.0. El objetivo es demostrar la viabilidad de una solución que combine resiliencia técnica, seguridad, trazabilidad y bajo coste, manteniendo el control siempre en manos del usuario.

Los resultados esperados incluyen un prototipo funcional que permita subir, compartir y recuperar ficheros entre nodos distribuidos de forma segura y trazable, así como evidencias de su robustez en escenarios de fallo y/o modificación de datos.

¹ GxP es un término genérico que agrupa distintas normativas de buenas prácticas aplicadas en sectores regulados, especialmente en la industria farmacéutica y biotecnológica. El acrónimo hace referencia a "Good x Practice", donde "x" representa áreas específicas como fabricación (GMP), laboratorio (GLP) o ensayos clínicos (GCP). Estas regulaciones buscan garantizar la calidad, trazabilidad e integridad de los procesos y productos.

² 21 CFR Part 11 es una regulación de la FDA (*Food and Drug Administration*) de Estados Unidos que establece los criterios bajo los cuales los registros y las firmas electrónicos se consideran confiables, equivalentes a los registros en papel y legalmente válidos en entornos regulados. Requiere control de acceso, trazabilidad y validación de los sistemas informáticos implicados.

1.3. Estructura del trabajo

El documento se estructura en varios apartados que desarrollan tanto los fundamentos teóricos como la implementación práctica de la solución:

1. El apartado “*Contexto y estado del arte*” presenta el entorno tecnológico en el que se sitúa el proyecto. Se analizan trabajos previos y soluciones existentes con el objetivo de identificar carencias y oportunidades que justifiquen la propuesta planteada.
2. En “*Objetivos y metodología de trabajo*” se exponen los objetivos que han guiado el desarrollo del proyecto. Se describe la estrategia adoptada, basada en una planificación por fases (análisis del problema, diseño del sistema, implementación, validación y evaluación).
3. “*Análisis de requisitos*” recoge los objetivos funcionales y no funcionales que guían el diseño. Se identifican las necesidades del usuario, las restricciones del entorno IoT y las características clave que debe cumplir la solución.
4. “*Diseño del sistema*” detalla la arquitectura propuesta. Se describen sus componentes, el modelado de datos y el flujo de operación. Se justifica el uso de las tecnologías elegidas y se establecen los principios de escalabilidad, seguridad y modularidad.
5. “*Implementación*” describe los aspectos técnicos del prototipo desarrollado: configuración de nodos IoT, estructura de microservicios, integración con *blockchain*, mecanismos de redundancia y recuperación ante fallos. Se incluyen fragmentos de código y esquemas.
6. “*Pruebas y validación*” presenta el entorno de pruebas, los escenarios simulados (como la pérdida de nodos, acceso simultáneo o verificación de integridad) y resultados obtenidos. Se evalúan parámetros como rendimiento, disponibilidad, eficiencia energética y resiliencia, comparándolos con los objetivos iniciales.
7. “*Conclusiones y trabajo futuro*” resume los principales logros alcanzados. Se reflexiona sobre las limitaciones del sistema y se proponen mejoras.

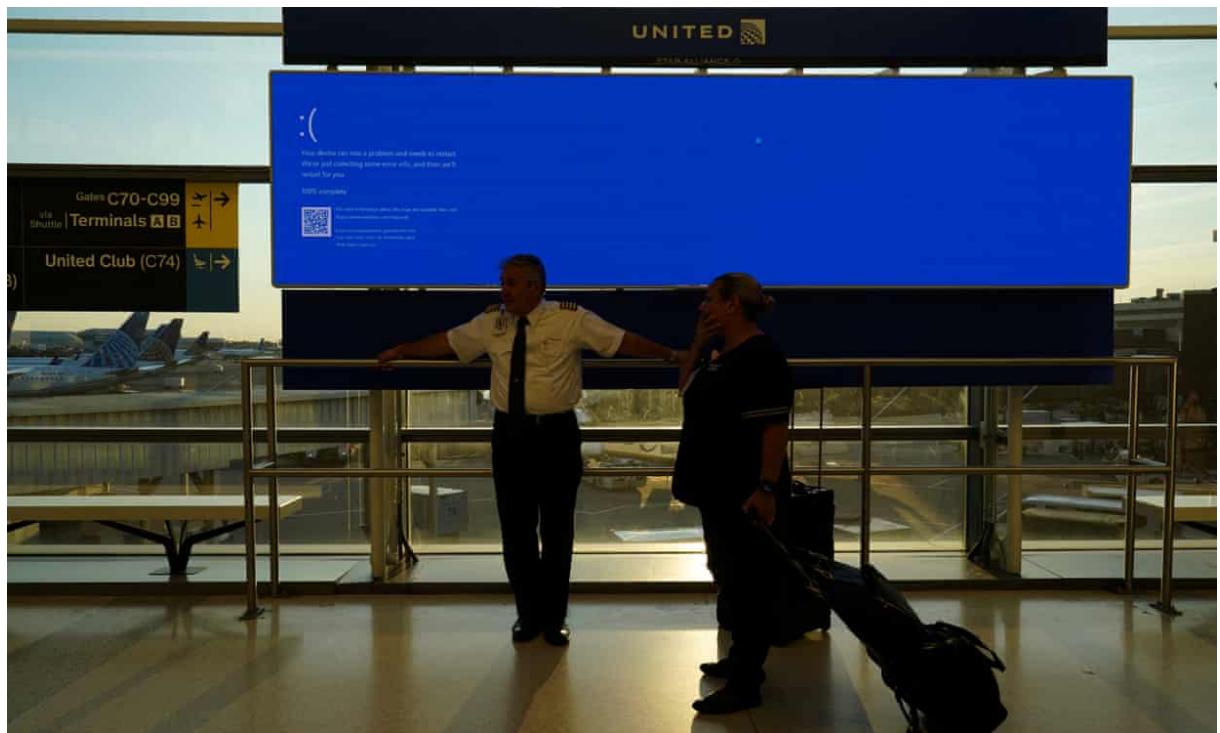
2. Contexto y estado del arte

Este apartado analiza el contexto actual del almacenamiento y la gestión descentralizada de datos. En la sección “*Estado del arte*” se revisan las soluciones existentes que fundamentan la propuesta.

2.1. Análisis del contexto

La transformación digital ha incrementado la demanda de soluciones capaces de gestionar datos de forma segura y eficiente. El modelo tradicional de almacenamiento en la nube presenta limitaciones en lo que respecta a la **soberanía del dato, resiliencia operativa y privacidad**. Las creciente concentración de servicios digitales en grandes corporaciones tecnológicas ha reavivado el interés por soluciones que devuelvan el control al usuario final (Whittaker , 2024).

Figura 2. Un fallo en Azure provoca el colapso de aeropuertos, hospitales y policía



Fuente: Pilkington & Aratani (2024).

La **Figura 2** ilustra el incidente del 19 de julio de 2024, donde una actualización defectuosa de *CrowdStrike* provocó interrupciones masivas a nivel mundial (Pilkington & Aratani , 2024). Es

un ejemplo que pone de manifiesto los riesgos inherentes a la centralización de servicios críticos en proveedores externos (por redundados que estén).

2.2. Estado del arte

Para contextualizar la propuesta, se establece un marco conceptual y tecnológico. Se revisan los fundamentos del IoT, los principios de *blockchain*, soluciones actuales de almacenamiento descentralizado y casos de éxito conocidos.

2.2.1. Introducción a la Web 3.0

A diferencia de la Web 2.0, caracterizada por la centralización del control de los datos en grandes plataformas, la Web 3.0 promueve una red donde la información, los servicios y las identidades digitales sean gestionados de forma descentralizada, transparente y verificable (Krause, 2023). La **Figura 3** ilustra sus rasgos más distintivos.

Figura 3. Principales características de la Web 3.0



Fuente: Anwar (2018).

Este nuevo paradigma se apoya en tecnologías como *blockchain* y contratos inteligentes, almacenamiento distribuido e identidad digital soberana³ (Buterin, 2014).

2.2.2. Redes Peer-to-Peer (P2P)

En entornos P2P, los nodos pueden ofrecer y solicitar servicios a la vez. Esta característica permite una mayor escalabilidad, tolerancia a fallos y resistencia a la censura. Tecnologías como *BitTorrent*, las primeras versiones de *Skype* y las redes *blockchain* públicas se basan en este modelo de arquitectura (Maymounkov & Mazieres, 2002).

Protocolos como IPFS (*InterPlanetary File System*) utilizan estructuras P2P para garantizar la integridad y el acceso descentralizado (Benet, 2014). Integran mecanismos de enrutamiento eficientes como las DHT (*Distributed Hash Tables*) y técnicas de verificación de integridad como *Merkle trees* para localizar y validar contenido mediante hashes criptográficos de forma segura y escalable (Stoica et al., 2001).

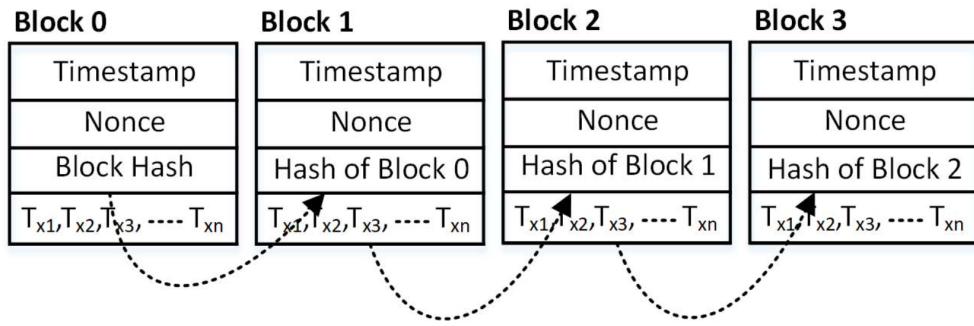
Para mejorar la tolerancia a fallos, en lo que respecta al almacenamiento de datos, utilizan técnicas de redundancia como la replicación, que consiste en mantener múltiples copias de un mismo fichero en diferentes nodos. Aunque simple y efectiva, implica un alto coste en almacenamiento (Benet, 2014). En su lugar, muchos sistemas modernos implementan *erasure coding*, que divide los datos en fragmentos y añade información redundante para su reconstrucción ante pérdidas parciales. Este enfoque es más eficiente que la replicación completa y se ha utilizado en soluciones como STORJ, donde se puede recuperar un archivo a partir de 29 de sus 80 fragmentos (Storj Labs, 2021).

2.2.3. Introducción a blockchain

En términos generales, *Blockchain* puede entenderse como un libro de registro digital descentralizado e inmutable, mantenido colectivamente por una red de nodos sin autoridad central (IBM, 2025).

³ La identidad digital soberana o SSI (*Self-Sovereign Identity*) es un modelo de gestión de identidad en el que los usuarios tienen pleno control sobre sus datos personales y permisos sin depender de autoridades centrales. Utiliza tecnologías como *blockchain* y criptografía para garantizar autenticidad, privacidad y verificabilidad de la información.

Figura 4. Estructura de bloques característica de una blockchain



Fuente: Ali et al., (2018, pág. 2).

El concepto ganó notoriedad con la aparición de *Bitcoin* en 2008, cuando Satoshi Nakamoto introdujo la *blockchain* como la base de un sistema de efectivo electrónico P2P (Nakamoto, 2008). Desde entonces, este tipo de tecnología ha evolucionado y se ha extendido a múltiples plataformas y aplicaciones, más allá de las criptomonedas originales.

La tecnología *Blockchain* se sustenta en cuatro principios clave (IBM, 2025): un registro distribuido que evita la dependencia de una autoridad central, la inmutabilidad de los datos registrados, un sistema de consenso descentralizado que valida las transacciones colectivamente y el uso de criptografía para garantizar la integridad y autenticidad de la información almacenada.

Blockchain aporta ventajas significativas frente a los sistemas tradicionales de gestión de datos: la descentralización del control, la transparencia de las transacciones, una seguridad reforzada mediante criptografía, consenso distribuido y trazabilidad completa que permite auditar con precisión el historial de cualquier operación o activo (IBM, 2025).

No obstante, a pesar de sus múltiples ventajas, la adopción de *Blockchain* también conlleva retos relevantes: escalabilidad limitada, alto consumo energético en ciertos modelos de consenso, latencia en la confirmación de transacciones y falta de un marco regulatorio claro. Estos factores deben evaluarse cuidadosamente, sobre todo en contextos que exigen alta disponibilidad, eficiencia energética y/o un cumplimiento normativo estricto (Mohammed Abdul, 2024).

2.2.3.1. Conceptos clave en blockchain

La comprensión de *Blockchain* implica estar familiarizado con una serie de conceptos básicos que pasamos a describir a continuación (IBM, 2025):

- **Bloques y cadena de bloques:** la *blockchain* se compone de una secuencia de bloques enlazados criptográficamente. Cada bloque incluye un conjunto de transacciones verificadas, una marca de tiempo y una referencia al bloque anterior. Su estructura lineal encadenada, asegura que cualquier cambio en un bloque afectaría también a los posteriores. La **Figura 4** muestra un ejemplo típico de estructura de bloques.
- **Transacciones:** una transacción representa una unidad de acción registrada en la red, como una transferencia de valor, un cambio de estado o la ejecución de un contrato inteligente. Antes de añadirse a un bloque, las transacciones se deben validar y firmar digitalmente para garantizar su autenticidad.
- **Hash criptográfico:** cada bloque está identificado por una huella digital generada mediante una función hash (SHA-256, MD5, etc.). El hash actúa como identificador único y asegura la inmutabilidad.
- **Nodo:** es cualquier dispositivo o entidad que participa en el mantenimiento y operación del sistema. Existen diferentes tipos de nodos según su función: los nodos completos almacenan toda la cadena de bloques y validan transacciones y bloques; los nodos ligeros conservan solo fragmentos de la cadena y dependen de los completos para operar; los nodos validadores o mineros son los encargados de crear nuevos bloques, de acuerdo con el mecanismo de consenso definido.
- **Mecanismo de consenso:** es el proceso por el que los nodos de la red acuerdan el estado actual de la cadena de bloques. La **Tabla 1** resume los mecanismos más habituales. Para más información, se puede consultar el anexo “[Mecanismos de consenso blockchain](#)”.
- **Contrato inteligente (Smart Contract):** programa que se ejecuta automáticamente en la *blockchain* cuando se cumplen ciertas condiciones. Fue popularizado por *Ethereum* y permite automatizar procesos complejos como pagos o acuerdos legales sin necesidad de intermediarios.

- **Billetera digital (Wallet)**: aplicación o dispositivo que permite al usuario gestionar sus claves criptográficas y operar con tokens o contratos inteligentes. No almacena los activos en sí, sino las claves necesarias para acceder a ellos.
- **Token y criptomonedas**: los tokens son activos digitales registrados en la *blockchain*. Pueden representar monedas nativas como BTC (*Bitcoin*) y ETH (*Ethereum*), *stablecoins*⁴, activos físicos o derechos digitales.
- **NFT (Non-Fungible Tokens)**: tokens no fungibles que representan activos digitales únicos e indivisibles registrados en la cadena de bloques. A diferencia de las criptomonedas tradicionales (que son fungibles e intercambiables entre sí), cada NFT tiene atributos irrepetibles. Se utilizan para certificar la autenticidad y la propiedad de activos como obras de arte, certificados o identificadores. Se implementan bajo estándares como ERC-721 o ERC-1155 en redes como *Ethereum*. Los NFT han ampliado las aplicaciones de *blockchain* hacia ámbitos como el arte digital, identidad soberana, trazabilidad de objetos físicos, representación de activos reales, etc.
- **DAPP (Decentralized Application)**: aplicación que funciona sobre una red de nodos descentralizados, habitualmente usando tecnología *blockchain*. A diferencia de las aplicaciones tradicionales, las DAPP no dependen de servidores centrales. Ejecutan su lógica de negocio mediante contratos inteligentes que suelen ser de código abierto.

Tabla 1. Principales mecanismos de consenso en blockchain

Mecanismo	Ejemplos	Eficiencia	Descentralización	Notas
Proof of Work (PoW)	Bitcoin, Litecoin	Baja	Alta	Alta seguridad, bajo rendimiento
Proof of Stake (PoS)	Ethereum, Cardano	Alta	Media-Alta	Escalable y eficiente, riesgo de concentración
Delegated PoS (DPoS)	EOS, TRON	Alta	Media	Rápido pero centralizado
Leased PoS (LPoS)	Waves	Alta	Media	Permite participación sin transferir propiedad

⁴ Una *stablecoin* es una criptomonedada cuyo valor está vinculado a un activo estable, como una moneda fiduciaria (ej. dólar, euro) o una materia prima (ej. oro), con el objetivo de reducir la volatilidad típica de otros criptoactivos.

Mecanismo	Ejemplos	Eficiencia	Descentralización	Notas
Proof of Authority (PoA)	VeChain, redes privadas	Muy alta	Baja	Adecuado para redes privadas
Proof of Burn (PoB)	Slimcoin	Media	Media	Destruye tokens para ganar derechos
Proof of Elapsed Time (PoET)	Hyperledger Sawtooth	Alta	Baja	Requiere hardware confiable
Proof of History (PoH)	Solana	Muy alta	Media	Optimiza el orden temporal
PBFT (Practical BFT)	Hyperledger Fabric	Alta	Baja	Muy eficiente en redes pequeñas
Proof of Space	Chia	Alta	Alta	Requiere espacio de almacenamiento
Hybrid PoW/PoS	Decred	Media	Media	Equilibrio entre seguridad y eficiencia

Fuente: elaboración propia.

2.2.3.2. Redes blockchain

Las redes *blockchain* se clasifican según el nivel de acceso y control (Campbell C. , 2025): abierto y descentralizado para redes públicas o controlado y eficiente para aplicaciones corporativas. La **Tabla 2** compara sus características, ventajas e inconvenientes. Para más información, se puede consultar el anexo “[Tipos de redes blockchain](#)”.

Tabla 2. *Tipos de redes en blockchain*

Tipo	Acceso	Validación	Transparencia	Ejemplos
Pública	Abierto a todos	Descentralizada (todos los nodos)	Total	Bitcoin, Ethereum
Privada	Restringido	Control centralizado	Limitada	Redes empresariales internas
Permisionada	Restringido según roles	Control distribuido con permisos	Dependiendo del diseño	Hyperledger Fabric, Corda
Híbrida	Mixto (público/privado)	Mixta: interna con validación pública parcial	Selectiva (segmentos públicos)	VeChain, IBM Food Trust

Fuente: elaboración propia.

La **Tabla 3** resume las redes más conocidas y sus características más relevantes. Para más información, se puede consultar el anexo [Principales Redes blockchain](#).

Tabla 3. Redes blockchain más extendidas

Red	Año de creación	Consenso	Contratos inteligentes	Transparencia	Token nativo
Bitcoin	2009	PoW	No	Total	BTC
Ethereum	2015	PoS (desde 2022)	Sí	Alta	ETH
Cardano	2017	Ouroboros PoS	Sí	Alta	ADA
Polkadot	2020	NPoS + BFT	Sí (en parachains)	Alta	DOT
Solana	2020	PoH + PoS	Sí	Moderada	SOL
Hyperledger Fabric	2015	Configurable (BFT, etc.)	Sí (privados, permisionados)	Limitada (permisionada)	-

Fuente: elaboración propia.

2.2.4. Introducción al almacenamiento distribuido

Los sistemas de almacenamiento distribuido surgen como respuesta a las limitaciones del almacenamiento centralizado tradicional. En los modelos distribuidos, los datos se fragmentan, replican y almacenan a través de múltiples nodos interconectados, proporcionando mayor **resiliencia, disponibilidad y escalabilidad**. La **Tabla 4** resume las principales diferencias entre el almacenamiento descentralizado y los modelos tradicionales:

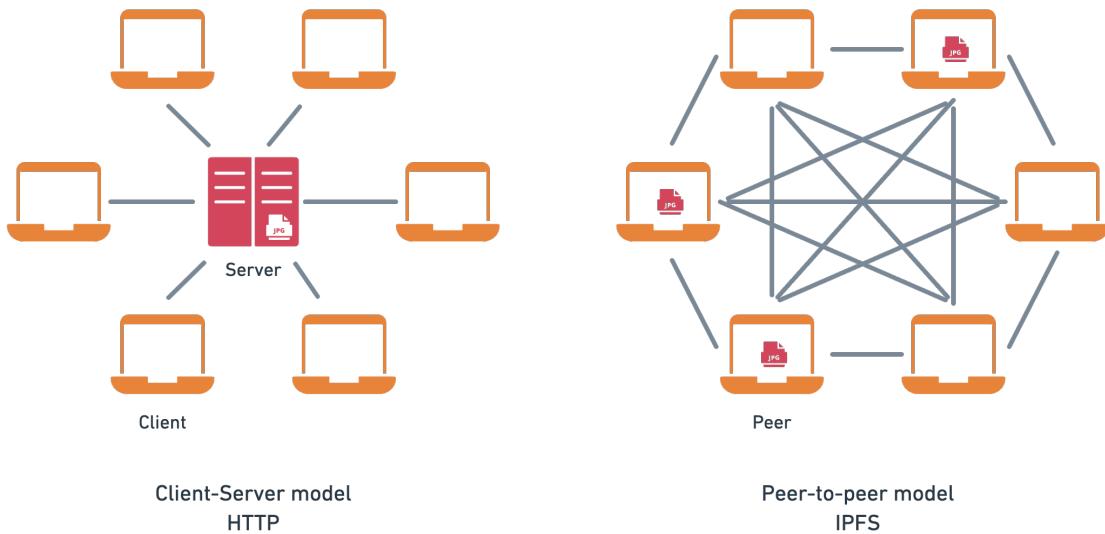
Tabla 4. Almacenamiento distribuido frente a centralizado o Cloud

Característica	Modelo Centralizado	Modelo Distribuido
Control de datos	Proveedor centralizado	Usuario / nodo propietario
Disponibilidad	Alta (infraestructura global)	Alta (depende de replicación y nodos activos)
Censura / alteración	Possible	Muy difícil
Coste	Variable, escalable	Generalmente menor, pero variable
Privacidad	Bajo control	Control criptográfico
Incentivos económicos	No aplicable	Basados en tokens o contratos
Dependencia del proveedor	Alta	Baja

Fuente: elaboración propia.

La **Figura 5** compara el modelo tradicional cliente-servidor con un enfoque *peer-to-peer*. Destaca cómo este último elimina el servidor central y permite la compartición directa de archivos entre nodos.

Figura 5. Arquitectura de IPFS frente a un modelo Cliente-Servidor tradicional



Fuente: Benet (2014).

2.2.4.1. IPFS (*InterPlanetary File System*)

IPFS es un protocolo y red P2P diseñado para almacenar y compartir datos de forma descentralizada. Reemplaza el modelo tradicional (*location-based addressing*⁵) basado en rutas por un sistema orientado al contenido (*content-addressed*), donde cada archivo se identifica por el hash de su contenido (Benet, 2014). Entre sus principales características destacan:

- **Arquitectura:** divide los archivos en bloques, los distribuye por nodos de la red y los enlaza mediante una estructura tipo *Merkle DAG*⁶. Al consultar un contenido, el sistema busca el nodo que posee el bloque correspondiente al hash solicitado.

⁵ *Location-based addressing* es un método tradicional de acceso a recursos en redes donde datos se solicitan en función de su ubicación (por ejemplo, una URL que apunta a un servidor específico) y no por su contenido.

⁶ *Merkle DAG* es una estructura de datos en forma de grafo acíclico dirigido (DAG) donde cada nodo contiene un hash criptográfico que depende de sus nodos hijos. Se utiliza para verificar la integridad de forma eficiente en sistemas distribuidos como IPFS o Git.

- **Funcionamiento:** opera con nodos voluntarios y un sistema de caché distribuido. Utiliza DHT⁷ para localizar los nodos que almacenan cada bloque.
- **Casos de uso:** distribución de contenido web, publicación científica, archivado digital, almacenamiento de datos en *blockchain*, ...
- Como **principales inconvenientes**, no garantiza la persistencia de los archivos a largo plazo (*pinning*⁸), no cifra la información de forma nativa y no incluye trazabilidad en sus operaciones.

2.2.4.2. Soluciones blockchain de almacenamiento distribuido

Con el auge de la tecnología *blockchain*, han aparecido nuevos sistemas de almacenamiento descentralizado que no solo distribuyen los datos, sino que también utilizan incentivos criptográficos y contratos inteligentes para garantizar la integridad, el acceso controlado y la retribución por el uso de recursos:

- **FILECOIN:** desarrollado por *Protocol Labs*, al igual que IPFS, *Filecoin* es una red *blockchain* que actúa como capa de incentivos para almacenamiento descentralizado. Los usuarios pagan por almacenar datos y los proveedores son recompensados con tokens por ofrecer espacio y cumplir pruebas de almacenamiento válidas (Protocol Labs, 2020). Entre sus características principales destacan su mecanismo de consenso combinado, basado en *Proof of Replication* y *Proof of Spacetime*, su integración nativa con IPFS que favorece la interoperabilidad y un enfoque orientado a la descentralización con un mercado dinámico de almacenamiento incentivado mediante criptomonedas.
- **STORJ:** utiliza una red de nodos voluntarios para alojar archivos cifrados y fragmentados, con retribución en su token STORJ. A diferencia de *Filecoin*, no tiene una *blockchain* propia, pero sí utiliza contratos inteligentes sobre *Ethereum* para gestionar los pagos (Storj Labs, 2021). Esta solución se caracteriza por utilizar cifrado

⁷ DHT o *Distributed Hash Table* es una estructura de almacenamiento distribuido que permite localizar datos de forma eficiente en redes P2P, asignando claves a nodos sin necesidad de un índice centralizado.

⁸ El *pinning* es un mecanismo en IPFS mediante el cual un nodo marca un archivo para conservarlo permanentemente en su almacenamiento local. Evita que sea eliminado por el recolector de basura del sistema. Es esencial para asegurar la disponibilidad del contenido a largo plazo.

de extremo a extremo y una estrategia de fragmentación basada en *erasure coding*, dividiendo los archivos en 80 fragmentos de los que solo se requieren 29 para su reconstrucción. Destaca por su facilidad de integración gracias a la compatibilidad con S3 y por ofrecer un rendimiento competitivo frente a alternativas centralizadas

- **SIA:** es una red de almacenamiento descentralizado con su propia *blockchain* y criptomoneda (SC, *Siacoin*). Los contratos entre usuarios y hosts se registran *on-chain*⁹ y los pagos se ejecutan tras la verificación de pruebas de almacenamiento (Vorick & Champine, 2014). Este sistema se basa en un mecanismo de consenso *Proof of Work* (PoW) combinado con contratos inteligentes, lo que permite alcanzar un alto grado de descentralización y un control criptográfico completo sobre los datos almacenados.
- **IAGON:** plataforma descentralizada que aprovecha la capacidad de almacenamiento y el poder de cómputo no utilizados de dispositivos en todo el mundo, creando una infraestructura de nube segura y eficiente. Al integrar tecnologías de *blockchain* e inteligencia artificial, IAGON permite a individuos y empresas contribuir con sus recursos excedentes y a cambio, recibir compensaciones en forma de tokens IAG. Emplea una arquitectura basada en fragmentación y cifrado de datos distribuidos entre múltiples nodos. Ha sido diseñado teniendo en cuenta el cumplimiento normativo, incluyendo regulaciones como el GDPR.

2.2.4.3. Ventajas e inconvenientes de los sistemas de almacenamiento distribuidos

Los sistemas de almacenamiento distribuidos y descentralizados representan una alternativa innovadora a los servicios tradicionales centralizados:

- **Resiliencia y tolerancia a fallos:** al distribuir los datos entre múltiples nodos independientes, se minimiza el riesgo de interrupciones por fallos únicos (Benet, 2014).
- **Mayor privacidad y soberanía:** los usuarios retienen el control sobre sus datos mediante cifrado y claves privadas, sin depender de terceros (Wilkinson et al., 2014).

⁹ *On-chain*: término que se refiere a las operaciones o datos que se registran directamente en una *blockchain*, haciéndolos públicos, inmutables y verificables por todos los participantes de la red.

- **Censura resistente:** al no haber una entidad central, los contenidos son difíciles de eliminar o bloquear una vez almacenados (Protocol Labs, 2020).
- **Incentivos económicos:** redes como *Filecoin* o *Sia* integran recompensas criptográficas para motivar la participación de proveedores de almacenamiento (Vorick & Champine, 2014).
- **Coste potencialmente reducido:** al aprovechar recursos infrautilizados, los precios pueden ser más competitivos en ciertos escenarios (Storj Labs, 2021).

Sin embargo, estas alternativas tampoco están exentas de limitaciones:

- **Complejidad técnica:** la gestión de claves, contratos inteligentes y nodos requiere conocimientos específicos, lo que limita su adopción generalizada.
- **Rendimiento variable:** la disponibilidad y velocidad de acceso dependen de factores como la ubicación y la fiabilidad de los nodos participantes.
- **Falta de garantías de servicio:** la ausencia de un proveedor central hace que los usuarios asuman mayor responsabilidad sobre la disponibilidad.
- **Costes indirectos por uso de blockchain:** en redes que integran contratos inteligentes, los *gas fees*¹⁰ pueden ser significativos (Zhang & Jacobsen, 2018).
- **Desafíos legales y regulatorios:** el almacenamiento distribuido plantea interrogantes respecto al cumplimiento de normativas como el RGPD, especialmente en cuanto a la ubicación de los datos y el derecho al olvido.

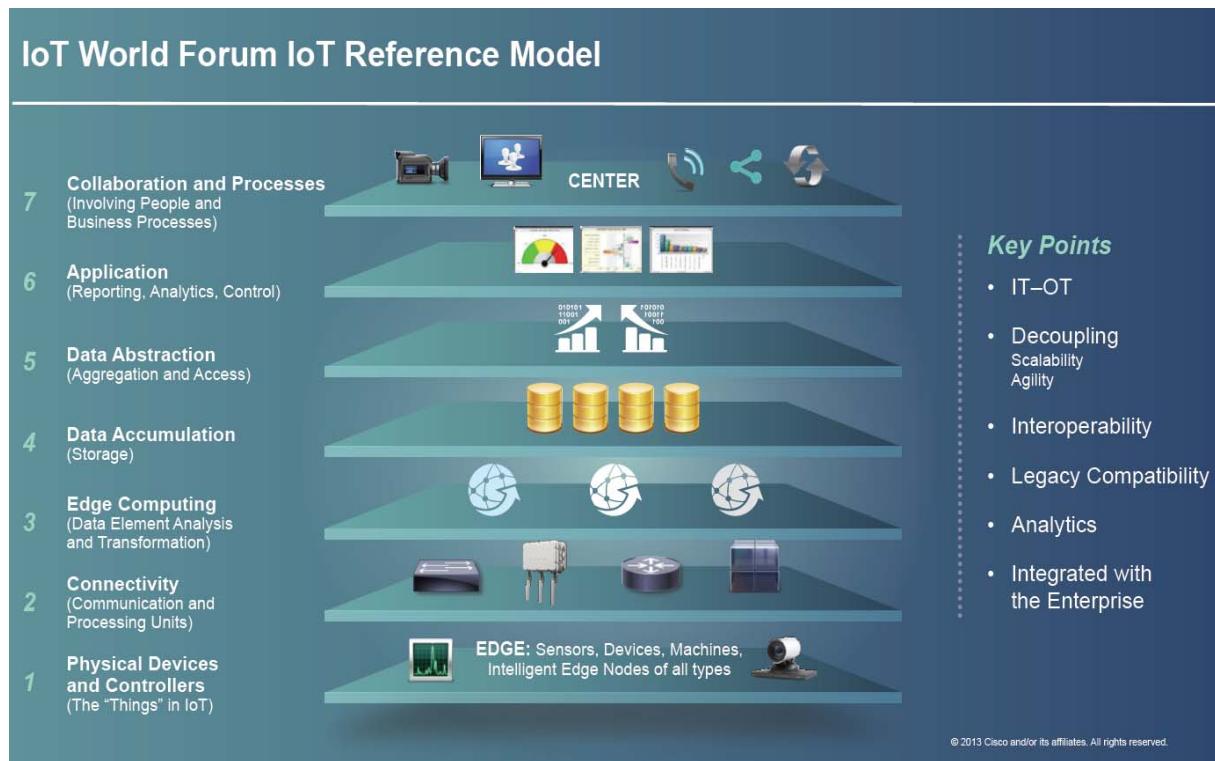
2.2.5. Introducción a IoT

El ecosistema IoT está compuesto por una serie de elementos interconectados que permiten la recolección, procesamiento, transmisión y explotación de datos provenientes del entorno físico. Entre sus principales componentes destacan los siguientes:

¹⁰ El *gas fee* es una comisión que se paga por ejecutar operaciones o transacciones en una red *blockchain* (especialmente en *Ethereum*). Se utiliza para compensar a los validadores por el uso de recursos computacionales y prevenir abusos del sistema.

- **Sensores:** encargados de captar variables físicas como temperatura, presión o luz y transformarlas en datos digitales.
- **Actuadores:** realizan acciones físicas en el entorno como abrir válvulas o encender dispositivos en respuesta a comandos.
- **Nodos:** dispositivos inteligentes que integran sensores, actuadores, capacidad de procesamiento y conectividad. Constituyen la "cosa" conectada en sí misma.

Figura 6. IoT World Forum Reference Model



Fuente: El Hakim (2018, pág. 2).

- **Pasarelas o gateways:** actúan como intermediarios entre los nodos IoT y la red externa (por ejemplo, Internet), gestionando la agregación y conversión de datos.
- **Backends:** almacenan, analizan y gestionan los datos recibidos, proporcionando servicios de analítica, visualización y control remoto.
- **Interfaces de usuario:** permiten al usuario visualizar la información y controlar los dispositivos de forma amigable.

- **Infraestructura de conectividad:** abarca tecnologías de red como Wifi, Bluetooth, Zigbee¹¹, LoRa¹² (*Long Range*), NB-IoT¹³ (*Narrowband IoT*). Adaptadas según contexto y necesidades del dispositivo.

La **Figura 6** presenta el modelo de referencia propuesto por el *IoT World Forum*. Estructura una solución IoT en siete niveles que van desde los dispositivos físicos a procesos de negocio.

2.2.5.1. Tecnologías clave en IoT

El ecosistema IoT se apoya en un conjunto de tecnologías que permiten la comunicación eficiente entre dispositivos, el procesamiento de datos y la interoperabilidad a gran escala:

- **Hardware embebido y dispositivos IoT** como microcontroladores (ej. ESP32, STM32), microordenadores (ej. *Raspberry Pi*, *Orange Pi*) y sensores de bajo consumo que forman la base física de las soluciones.
- **Computación en el borde (edge) y en la niebla (fog)**. Paradigmas que permiten procesar datos cerca de la fuente para reducir la latencia y minimizar el tráfico.
- **Plataformas en la nube**. Ofrecen servicios escalables de almacenamiento, análisis de datos, visualización y gestión remota de dispositivos (ej. AWS IoT, Azure IoT Hub).
- **Protocolos de comunicación** especializados como MQTT¹⁴ (*Message Queuing Telemetry Transport*) y CoAP¹⁵ (*Constrained Application Protocol*), diseñados para entornos con recursos limitados y transmisión eficiente de datos en redes de sensores.

¹¹ Zigbee es un protocolo de comunicación inalámbrica de bajo consumo y corto alcance basado en estándares IEEE 802.15.4. Ideal para redes malladas de dispositivos IoT en aplicaciones como domótica e industria.

¹² LoRa o *Long Range* es una tecnología de comunicación inalámbrica de baja potencia y largo alcance, diseñada para redes IoT, especialmente para entornos rurales o urbanos con dispositivos dispersos de bajo consumo energético.

¹³ NB-IoT o *Narrowband IoT* es una tecnología de red celular estandarizada para IoT, optimizada para ofrecer cobertura amplia, bajo consumo energético y alta penetración en interiores. Ideal para dispositivos que transmiten pequeños volúmenes de datos de forma ocasional.

¹⁴ MQTT o *Message Queuing Telemetry Transport* es un protocolo ligero de mensajería tipo *publish / subscribe*, diseñado para comunicaciones eficientes y fiables en redes con recursos limitados.

¹⁵ CoAP o *Constrained Application Protocol* es un protocolo ligero basado en REST diseñado para dispositivos con recursos limitados que permite comunicación eficiente mediante el modelo cliente-servidor sobre UDP.

- **Tecnologías de conectividad**, que incluyen opciones de corto alcance como BLE¹⁶ (*Bluetooth Low Energy*) o Zigbee, largo alcance como LoRaWAN¹⁷ (*Long Range Wide Area Network*) o Sigfox¹⁸ y redes celulares específicas para IoT como NB-IoT o LTE-M¹⁹ (*Long Term Evolution for Machines*).
- **Inteligencia artificial y analítica de datos**. Convierten los datos en conocimiento, permitiendo desde el mantenimiento predictivo hasta la automatización inteligente.
- **Ciberseguridad y privacidad** con mecanismos específicos para proteger dispositivos y datos en entornos distribuidos heterogéneos: cifrado ligero, autenticación segura, cumplimiento normativo, etc.

2.2.5.2. Arquitecturas típicas en IoT

Las arquitecturas de sistemas IoT se estructuran comúnmente en tres niveles jerárquicos: nube (*cloud*), borde (*edge*) y niebla (*fog*). Tradicionalmente ha dominado el modelo centrado en la nube, donde los dispositivos envían todos los datos a servidores remotos para su procesamiento y almacenamiento. Si bien esta arquitectura es escalable, presenta limitaciones en términos de latencia, dependencia de conectividad y uso de ancho de banda. Como respuesta, han surgido paradigmas complementarios que distribuyen la carga de trabajo de forma más eficiente, mejorando la resiliencia, privacidad y escalabilidad del sistema IoT (Yousefpou et al., 2018).

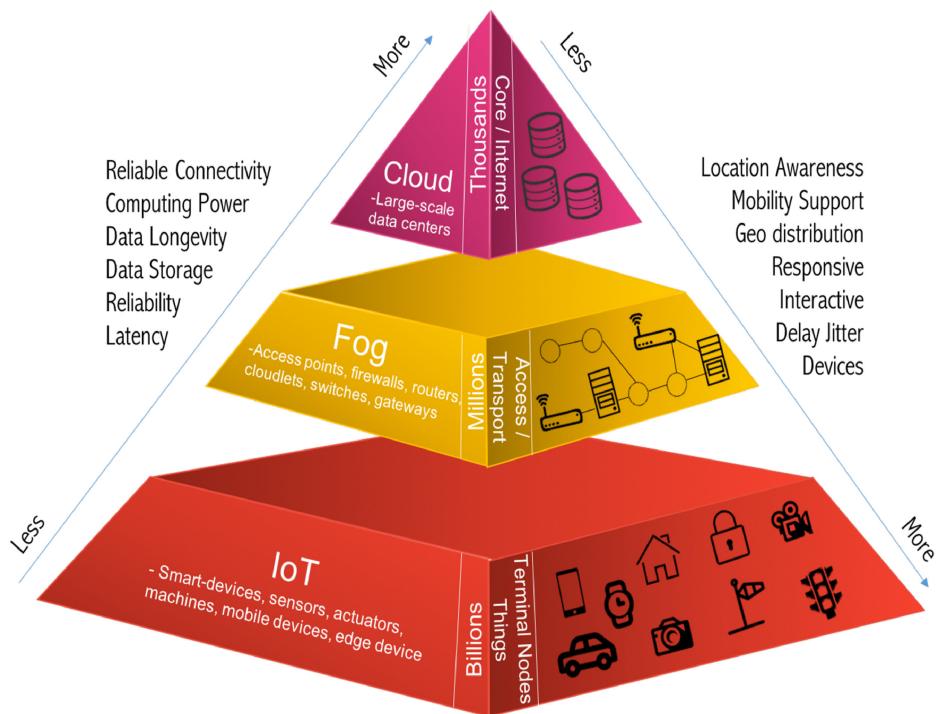
¹⁶ BLE o *Bluetooth Low Energy* es una tecnología inalámbrica de corto alcance optimizada para dispositivos IoT que permite comunicaciones con bajo consumo energético y tiempos de respuesta rápidos, ideal para sensores y wearables.

¹⁷ LoRaWAN o *Long Range Wide Area Network* es un protocolo de red LPWAN diseñado para comunicaciones inalámbricas de largo alcance y bajo consumo energético, utilizado en aplicaciones IoT distribuidas como agricultura, ciudades inteligentes o monitoreo ambiental.

¹⁸ Sigfox es una tecnología LPWAN propietaria orientada a comunicaciones IoT de largo alcance y ultra bajo consumo, ideal para transmitir pequeños volúmenes de datos de forma esporádica en aplicaciones como seguimiento de activos y sensores remotos.

¹⁹ LTE-M o *Long Term Evolution for Machines* es una tecnología de red celular IoT que ofrece mayor ancho de banda y menor latencia que NB-IoT, manteniendo bajo consumo energético. Ideal para aplicaciones que requieren movilidad, voz y transmisión de datos moderada.

Figura 7. Arquitectura de tres capas en fog Computing



Fuente: Yousefpou et al. (2018, pág. 302).

La **Figura 7** muestra el modelo jerárquico *Cloud–Fog–IoT*: mientras la capa IoT agrupa los dispositivos inteligentes, la capa en la niebla actúa como intermediaria local con capacidad de procesamiento distribuido y la nube centraliza, entre otras cosas, el almacenamiento masivo.

2.2.6. IoT y blockchain

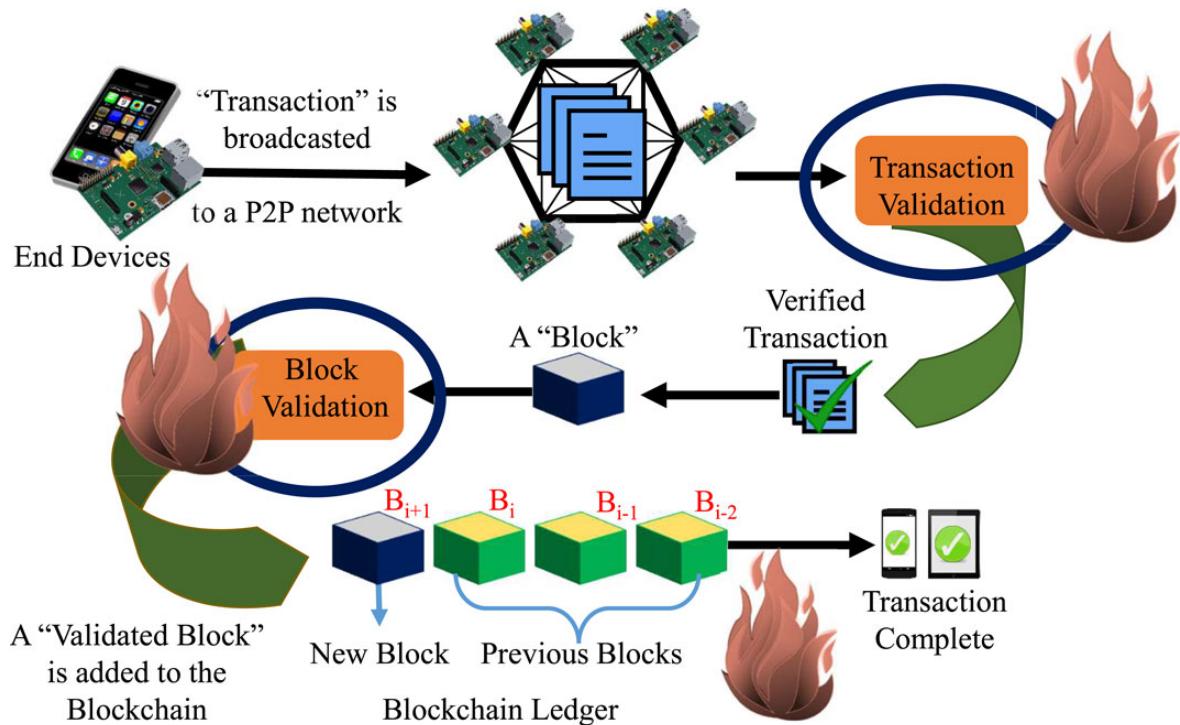
Al combinar IoT con *blockchain*, se busca una fuente de datos descentralizada, transparente e inmutable que permita mejorar la trazabilidad de los activos físicos, la seguridad de los datos y la confianza entre múltiples participantes sin necesidad de servidores centrales (Chen et al., 2023).

No obstante, integrar *blockchain* en entornos IoT presenta desafíos significativos. Las redes públicas tradicionales como *Bitcoin* o *Ethereum* requieren de altos recursos de cómputo, almacenamiento local y consumo energético para participar plenamente.

La **Figura 8** muestra una operación típica de validación en una *blockchain*. Aparecen representadas las operaciones más costosas desde un punto de vista energético. Contrastá con los dispositivos IoT típicos de bajo costo y consumo, que envían datos a un servicio central

sin mucha seguridad adicional (Boyce, 2019). Manejar una *blockchain* tradicional completa en dispositivos IoT limitados es inviable en la práctica (Bapatla et al., 2023).

Figura 8. Generación de bloques, validación y adición en blockchain



Fuente: Bapatla et al. (2023, pág. 3).

Por otro lado, usar una cadena de bloques para almacenar directamente grandes volúmenes de datos IoT tampoco es práctico en términos de costo y latencia (Boyce, 2019). Surgen nuevas tecnologías y enfoques híbridos para adaptar el almacenamiento distribuido y la *blockchain* a las restricciones IoT.

2.2.7. Enfoques de almacenamiento distribuido para IoT

Se discuten varios enfoques de almacenamiento distribuido que combinan redes descentralizadas con servicios tradicionales, equilibrando rendimiento, coste y trazabilidad.

2.2.7.1. Arquitecturas híbridas

Una estrategia común es el empleo de **arquitecturas híbridas**. La carga se reparte entre el dispositivo, la nube / fog y los nodos intermedios. En lugar de que cada sensor ejecute un nodo

completo, se suele emplear un gateway o servidor de borde que actúa de intermediario. Un patrón típico es que los dispositivos IoT envíen sus datos a través de protocolos ligeros (ej. MQTT) a un gateway local y que este se encargue de procesar y transmitir la información a la *blockchain*. De este modo, se centraliza el manejo “pesado” de *blockchain*, manteniendo una carga razonable en los sensores. Aunque sacrifica parte de la descentralización, puede ser un compromiso práctico en entornos industriales.

Como ejemplo ilustrativo, un experimento integró un lector de códigos de barras IoT con Ethereum a través de un servidor Python (*Flask + Web3.py*) y demostró la viabilidad técnica de anclar transacciones desde IoT a *blockchain*. Concluyó que es más sencillo usar la *blockchain* para “anclar” los datos recogidos (ej. usando hashes), en lugar de que cada sensor interactúe directamente con la cadena de bloques (Boyce, 2019).

2.2.7.2. Almacenamiento distribuido off-chain

Otra pieza clave es el uso de **almacenamiento distribuido off-chain**. Dado que las *blockchains* no son eficientes manejando grandes cantidades de datos crudos, se emplean redes P2P como IPFS (*InterPlanetary File System*). En una arquitectura típica, los datos IoT se guardan en nodos distribuidos de IPFS (u otra base de datos descentralizada). En la *blockchain* solo se registra su hash para asegurar la inmutabilidad. Así se consigue un sistema *serverless*²⁰ para dispositivos de bajo rendimiento sin punto único de fallo.

Como ejemplo, se implementó un sistema de monitorización industrial donde una *Raspberry Pi*, como gateway LoRaWAN, publicaba datos de sensores en IPFS. Mientras una *blockchain* aportaba la confianza e integridad necesarias, se garantizaba una fuente de datos resiliente y accesible tanto local como remotamente. Incluso nodos muy modestos pudieron participar en estas pruebas (Froiz-Míguez et al., 2019). El experimento demostró que se pueden combinar almacenamiento distribuido off-chain con *blockchain*, permitiendo a IoT escalar sin saturar la cadena de bloques.

²⁰ En el ámbito de *blockchain*, el término *serverless* hace referencia a aplicaciones cuya lógica se ejecuta de forma descentralizada, sin depender de servidores tradicionales. En lugar de infraestructura gestionada centralmente, los contratos inteligentes y los nodos de la red asumen el rol de *backend*, lo que elimina puntos únicos de fallo y reduce la necesidad de servidores propios.

2.2.8. Tecnologías blockchain ligeras para IoT

Debido a las limitaciones mencionadas, han surgido tecnologías DLT²¹ (*Distributed Ledger Technology*) diseñadas específicamente para entornos de bajo consumo como IoT. Estas plataformas priorizan transacciones ligeras, ausencia de comisiones y bajos requerimientos de hardware.

2.2.8.1. IOTA (Tangle)

IOTA es uno de los **proyectos pioneros enfocados a IoT**. A diferencia de una *blockchain* tradicional, IOTA utiliza una arquitectura DAG²² (*directed acyclic graph*) llamada *Tangle*. No hay bloques ni mineros: cada dispositivo que envía una transacción debe validar a su vez dos transacciones previas. Este mecanismo elimina las comisiones por transacción y hacen viable el intercambio de microdatos / micropagos M2M²³ (*Machine to Machine*) sin coste (Hamilton, 2024).

IOTA provee una infraestructura IoT descentralizada y *trustless*²⁴ donde los dispositivos pueden compartir sus datos de forma directa y segura. Su token nativo MIOTA, sirve para acceder a ciertas funciones (ej. contratos inteligentes en las versiones nuevas de la red). Tras enfrentar desafíos de seguridad en sus inicios, como el ataque que sufrió en 2018²⁵, IOTA ha mejorado su protocolo (Hamilton, 2024).

²¹ DLT o *Distributed Ledger Technology* hace referencia en general a tecnologías que permiten el registro de datos de forma distribuida, como *blockchain*.

²² Un DAG, *Directed Acyclic Graph* o *Grafo Dirigido Acíclico* es una estructura de datos utilizada como alternativa a la cadena de bloques tradicional en sistemas de registro distribuido. A diferencia de *blockchain*, que agrupa las transacciones en bloques secuenciales, un DAG permite que cada transacción se conecte directamente con otras anteriores, formando una red sin ciclos que facilita la validación paralela. Esta arquitectura mejora la escalabilidad, reduce la latencia y elimina la necesidad de minería, por lo que resulta especialmente adecuada para entornos como el IoT y redes con dispositivos de bajo consumo.

²³ M2M o *Machine to Machine* se refiere a la comunicación directa entre dispositivos sin intervención humana, permitiendo el intercambio automático de datos entre sensores, actuadores u otros sistemas electrónicos. Comúnmente utilizada en entornos IoT e industriales.

²⁴ *Trustless* se refiere a un sistema donde la confianza se sustituye por mecanismos técnicos y descentralizados que aseguran el cumplimiento de las reglas, incluso entre partes que no se conocen ni se fían entre sí.

²⁵ En 2018, miles de usuarios de IOTA fueron víctimas de un robo masivo tras utilizar un generador de claves online comprometido. Aunque el protocolo no fue vulnerado, el atacante recopiló las semillas y accedió a los fondos una vez que las carteras fueron financiadas. El caso evidenció la importancia de generar claves de forma segura y local, sin depender de herramientas no auditadas.

En el contexto IoT, se ha probado con casos de uso como ciudades inteligentes, automoción y trazabilidad industrial. Su capacidad de registrar datos inmutables sin fees, lo hacen atractivo para registrar en tiempo real eventos en la cadena de suministro. Existen librerías y APIs que facilitan la integración de dispositivos IoT, permitiendo enviar datos a la *Tangle* de forma sencilla. IOTA representa una alternativa ligera a las *blockchains* convencionales y está pensada desde cero para ecosistemas IoT.

2.2.8.2. Nano

Nano es una **criptomoneda ultraligera que también elimina la minería y las comisiones**, ganando atención por su eficiencia energética (Nano, 2025). Emplea una arquitectura de tipo *block-lattice*²⁶ y un mecanismo de consenso por votación ORV²⁷ (*Open Representative Voting*) en lugar de PoW o PoS. Gracias a estas características, procesar transacciones consume menos energía (órdenes de magnitud) que en otras redes y los requisitos de hardware para participar son mínimos. Se ha calculado que una transacción de Nano puede usar tan solo 0.111 W/h, haciendo que sea un candidato atractivo para dispositivos IoT que necesiten enviar micro pagos y/o datos frecuentemente, sin agotar baterías ni ancho de banda.

La ausencia de tarifas facilita su escalabilidad en escenarios M2M. Si bien *Nano* se diseñó principalmente como moneda digital, su filosofía verde y de infraestructura minimalista han llevado a explorar su uso en entornos IoT. Los desarrolladores pueden interactuar con Nano mediante APIs RPC o librerías, usando múltiples lenguajes.

2.2.8.3. Helium

Helium es una plataforma que une *blockchain* con las redes inalámbricas IoT. A diferencia de IOTA y Nano, más centradas en transacciones de datos / valor, ***Helium se enfoca en crear una***

²⁶ El *block-lattice* es una arquitectura donde cada cuenta gestiona su propia cadena de bloques, lo que permite una ejecución extremadamente rápida, sin necesidad de agrupar transacciones en bloques globales. Esta estructura innovadora elimina la congestión y reduce la necesidad de consenso centralizado, haciendo que la red sea ligera y eficiente.

²⁷ ORV u *Open Representative Voting* es un sistema de votación descentralizado. Permite a los usuarios de la red delegar el poder de voto a representantes para validar transacciones y mantener el consenso, sin necesidad de minería ni recompensas monetarias. Es eficiente, rápido y energéticamente sostenible.

infraestructura inalámbrica descentralizada mediante incentivos *blockchain* (Hamilton, 2024).

Permite desplegar nodos físicos (*hotspots*) que actúan como gateways *LoRaWAN* de largo alcance y bajo consumo, recompensando a sus dueños con tokens HNT por proveer de cobertura a los dispositivos IoT cercanos. Es, esencialmente, una red comunitaria impulsada por *blockchain*: los sensores IoT transmiten vía radio (*LoRa*) a cualquier *hotspot* disponible, la red registra esas transmisiones y distribuye recompensas. Gracias a este enfoque se construyó en pocos años una de las mayores redes IoT LPWAN²⁸ (*Low-Power Wide-Area Network*) existentes. Útil para aplicaciones como rastreo de activos, agricultura o ciudades inteligentes, sin depender de operadores celulares tradicionales. Inicialmente operó su propia *blockchain*, pero en 2023 migró a *Solana*, buscando mayor rendimiento y capacidad de *smart contracts*. Los *hotspots* siguen usando un mecanismo de PoC²⁹ (*Proof-of-Coverage*) para validar que brindan servicio real a dispositivos IoT.

Para los desarrolladores, ofrece APIs para enviar datos (*Helium Console*) y un SDK para integración de conectividad en aplicaciones. Desde *Python*, un dispositivo como *Raspberry Pi* podría usar librerías *LoRa* para comunicarse, mientras que las aplicaciones de servidor podrían consultar el *blockchain* de *Helium* (ahora en *Solana*) para verificar y utilizar los datos transmitidos.

Helium es un caso singular de *blockchain* para IoT: en vez de asegurar datos de sensores en una *blockchain*, usa la *blockchain* para motivar la creación de una red inalámbrica global IoT de bajo consumo, demostrando un uso innovador de dicha tecnología.

2.2.8.4. Otras plataformas y frameworks

Además de las plataformas mencionadas, existen otras iniciativas notables. Por ejemplo, ***IoTeX* es una plataforma *blockchain* abierta orientada a IoT que destaca por ser compatible**

²⁸ LPWAN o *Low-Power Wide-Area Network* es un tipo de tecnología de red inalámbrica diseñada para comunicar dispositivos a larga distancia con un consumo energético muy bajo. Ideal para aplicaciones IoT que requieren transmitir pequeñas cantidades de datos de forma esporádica y eficiente.

²⁹ PoC o *Proof-of-Coverage* es un algoritmo de consenso que verifica que los nodos en una red descentralizada estén proporcionando cobertura de red en el mundo físico, usando desafíos y testigos entre dispositivos. Se usa principalmente en redes IoT como *Helium* para incentivar la expansión geográfica real de una red.

con la EVM³⁰ (*Ethereum Virtual Machine*). Esto facilita a los desarrolladores portar contratos inteligentes y *dApps* de *Ethereum* a entornos IoT basados en *IoTeX*, aprovechando su enfoque de privacidad y escalabilidad (Hamilton, 2024).

Por otro lado, ***VeChain*** se ha posicionado en la trazabilidad de la cadena de suministro, proporcionando una *blockchain* pública-permisionada que se ha usado para registrar el recorrido y autenticidad de los productos. Desde 2016, *VeChain* incorporó características como soporte nativo a IoT (RFID, NFC, sensores) y contratos inteligentes para gestionar activos físicos (Hamilton, 2024). En el sector farmacéutico, *VeChain* ha liderado proyectos con varios fabricantes para asegurar la procedencia de sus medicamentos (DrugPatentWatch, 2024).

En el ámbito de código abierto, los frameworks de *Hyperledger* merecen especial atención. Aunque *Hyperledger Fabric* o *Sawtooth* son demasiado pesados para correr directamente en dispositivos pequeños, sí se han utilizado como base para soluciones industriales IoT. *Fabric*, ha sido la columna vertebral de plataformas de IBM para trazabilidad alimentaria y farmacéutica: los dispositivos IoT proveen los datos y el consenso corre en la nube (DrugPatentWatch, 2024).

***BigchainDB*, combina una base de datos distribuida con propiedades de *blockchain*.** Se ha explorado para almacenar registros IoT a gran velocidad, delegando la confianza a mecanismos BFT (*Byzantine Fault Tolerance*).

A nivel académico, se proponen nuevos algoritmos de consenso ligeros adaptados a IoT. Un ejemplo es ***EasyChain*, una blockchain experimental que introduce un consenso PoAh (*Proof-of-Authentication*) simplificado** (Bapatla et al., 2022). *EasyChain* fue implementada en *Python*, buscando facilitar su integración con aplicaciones IoT. Desplegada sobre un sistema de placa reducida logró latencias de 150 ms por transacción, demostrando su viabilidad en entornos de recursos limitados. Este tipo de investigaciones sugiere que futuras redes podrán ejecutarse parcialmente en el borde (*edge*) con dispositivos modestos, mediante criptografía y consensos optimizados de bajo consumo.

³⁰ EVM o *Ethereum Virtual Machine* es el entorno que permite ejecutar contratos inteligentes en la red *Ethereum*. Actúa como una máquina virtual descentralizada, determinista y aislada, garantizando que el comportamiento de los programas sea coherente en todos los nodos de la red.

2.2.9. Ejemplos de casos de uso

Para ilustrar la aplicabilidad real de las tecnologías analizadas, se presentan algunos casos de uso representativos que demuestran cómo *blockchain* y el almacenamiento distribuido pueden resolver problemas concretos en diferentes sectores.

2.2.9.1. Almacenamiento de contenido web en blockchain

En 2023, el ecosistema de *Cardano* logró un hito significativo al alojar el primer sitio web estático en su plataforma (Dave, 2023). Este avance demuestra la capacidad de *Cardano* para almacenar y servir contenido web directamente desde su *blockchain* sin depender de servidores centralizados (Cardano Spot, s.f.).

Figura 9: Post en X del creador de la primera web hospedada en Cardano

The first static Website hosted & served on Cardano....

This is a successful proof of concept test I have carried out utilising IAGON decentralised storage. 🎉

Just replicated my theory of Decentralised Static Website Hosting on top of IAGON decentralised storage infrastructure. 🚀

This will be the **FIRST EVER** website served from decentralised storage on Cardano to my knowledge (maybe the first across all blockchains I don't know) Of course this will open up further options and is just a proof of concept around possibilities. 🌟🚀

What I did... I replicated and cloned the IAGON Website as a static website (had to tweak as didn't have source code for this & it uses the Next.js framework), I uploaded & hosted this up to the lagon storage dashboard meaning this was now hosted on decentralised Infrastructure that is offered by individual storage providers. 🚀 I then took this a step further and added javascript dependencies to prove also a Next.js framework website can be used & functional.

Big thanks to the IAGON team, for assisting & doing the development work required to facilitate a prove this is achievable on their platform, it's a testament to the team on achieving this in just a few weeks whilst pushing for mainnet milestones, really pushing the boundaries of what's possible. ❤️

Fuente: Dave (2023).

La **Figura 9** ilustra el post donde el autor hace público su experimento de alojamiento web en la red de *Cardano*. La prueba de concepto demuestra la viabilidad de modelos nuevos de almacenamiento.

2.2.9.2. PingER: Almacenamiento y acceso descentralizado con blockchain

El proyecto PingER (*Ping End-to-End Reporting*) es una iniciativa global para la medición del rendimiento de Internet entre distintos puntos del mundo. Tradicionalmente, su arquitectura se ha apoyado en un modelo centralizado gestionado por el SLAC³¹ (*Stanford Linear Accelerator Center*) que centraliza la recopilación, almacenamiento y análisis de los datos generados por más de 50 agentes de monitoreo distribuidos globalmente.

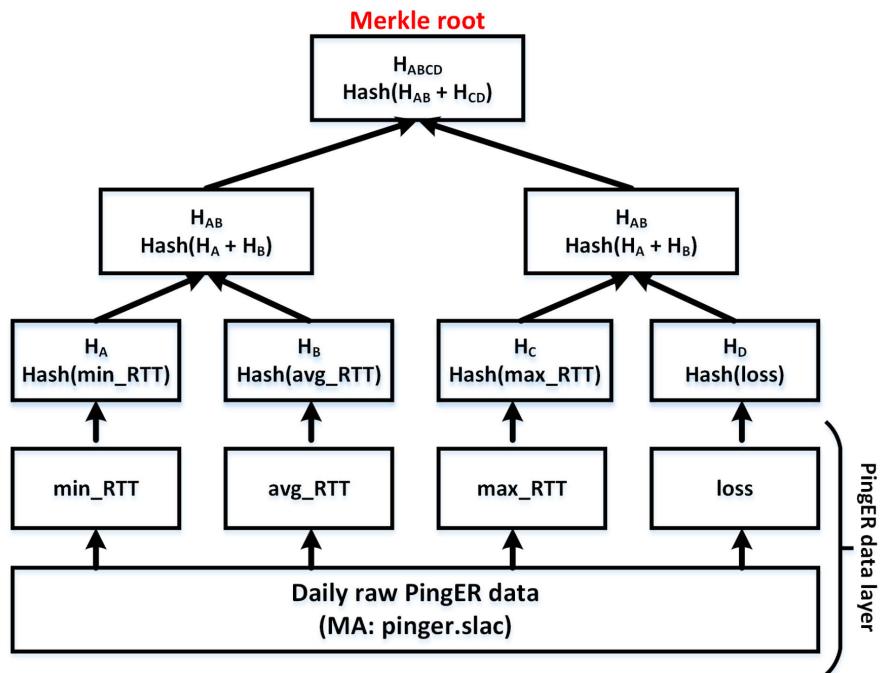
Para superar las limitaciones de este enfoque, los autores proponen una arquitectura basada en *blockchain* con permisos (Ali et al., 2018). Esta transformación convierte a cada agente de monitoreo en un nodo dentro de una red P2P:

- Los metadatos de los archivos de medición se almacenan en la *blockchain*.
- Los datos reales se almacenan *off-chain*, utilizando una red P2P con DHT y técnicas de *erasure coding*, lo que mejora la disponibilidad y tolerancia a fallos.
- Se utiliza una estructura *Merkle tree*³² para verificar la integridad de los archivos y asegurar la inmutabilidad de los datos.
- El consenso en la red se logra mediante el algoritmo *Simplified Byzantine Fault Tolerance* (SBFT).
- El acceso a los datos se mantiene compatible con los scripts existentes mediante direccionamiento por hash, incluso a través de IPFS.

³¹ El SLAC o *Stanford Linear Accelerator Center* es un centro de investigación operado por la *Universidad de Stanford* que alberga uno de los aceleradores lineales de partículas más largos del mundo, dedicado a la física de partículas, materiales y láseres de rayos X.

³² Un *árbol de Merkle* o *Merkle tree* es una estructura de datos en forma de árbol binario que permite verificar de forma eficiente y segura la integridad de grandes volúmenes de datos. Cada nodo hoja contiene el hash de un bloque de datos, y cada nodo intermedio es el hash de la concatenación de sus nodos hijos, hasta llegar a la raíz (*Merkle root*), que resume todo el contenido. La **Figura 10** ilustra su estructura de datos.

Figura 10. Estructura de datos Merkle tree



Fuente: Ali et al. (2018, pág. 4).

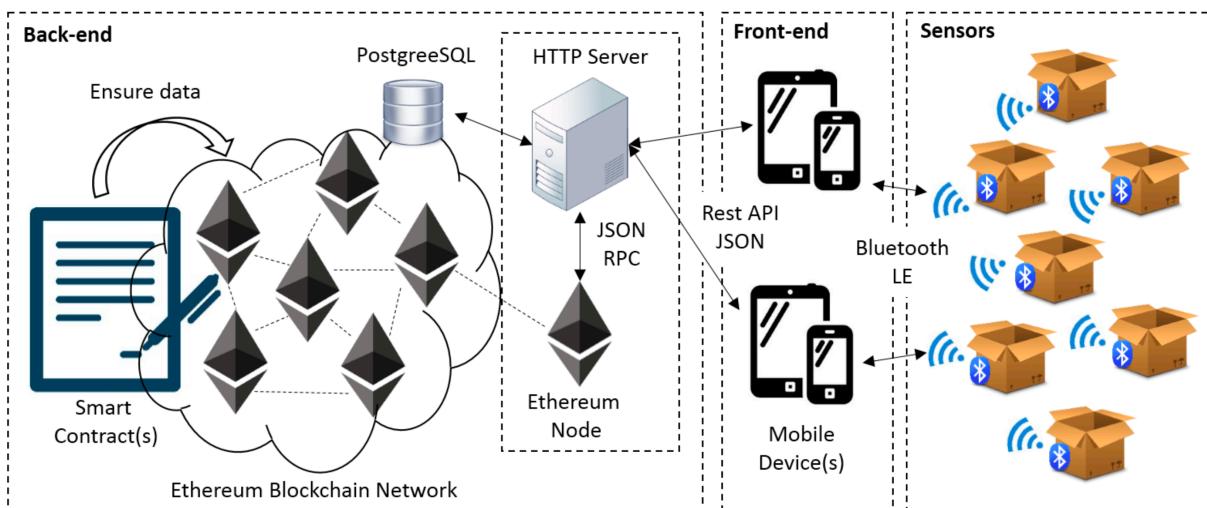
2.2.9.3. Trazabilidad en la cadena de suministro farmacéutica

Uno de los campos más prometedores para la convergencia IoT-blockchain es la **trazabilidad de la cadena de suministro**. Es de especial interés en industrias como la farmacéutica donde la autenticidad y condiciones de los productos son críticos. La combinación de sensores IoT con una red *blockchain*, permite crear un registro inviolable del recorrido realizado por un medicamento: desde su fabricación hasta el paciente, incluyendo condiciones ambientales, transferencias de custodia, etc. Aborda desafíos de transparencia y visibilidad: cada participante puede tener acceso en tiempo real a un historial inmutable de cada lote o ítem, lo que ayuda a prevenir falsificaciones y fraudes en la cadena (DrugPatentWatch, 2024). A continuación, se muestran algunos casos de éxito representativos:

- **Monitoreo de cadena de frío:** *Modum* es una startup suiza que demostró un caso de uso donde sensores de temperatura IoT, viajaban con los lotes de medicamentos sensibles y registraban continuamente la temperatura durante el transporte. Esos datos, se almacenaban en la *blockchain* de *Ethereum*, vinculados a cada envío. Contratos inteligentes evaluaban automáticamente si la temperatura se mantenía

dentro de rango seguro, notificando a las partes involucradas (Bocek et al., 2017). De este modo se garantizaba la integridad de los registros, que eran auditables. El sistema redujo los costos operativos, comparado con el proceso tradicional y aseguró el cumplimiento de la normativa de buenas prácticas de distribución (GDP). *Modum* logró usar *Ethereum* como infraestructura para contratos autoejecutables que validaban las condiciones de transporte y aportaban evidencias *tamper-proof*³³ de la calidad del producto entregado. Posteriormente se han implementado soluciones inspiradas en *Modum* que, adquirida por UPS, continúa desarrollando registradores de temperatura con *blockchain*. La propuesta ha ganado varios premios por mejorar la eficiencia logística (Campbell R. , 2016). La **Figura 11** muestra la arquitectura de *Modum*.

Figura 11. Arquitectura blockchain de Modum.io AG



Fuente: Bocek et al. (2017, pág. 4).

- **Autenticidad de medicamentos:** la lucha contra medicamentos falsificados es prioritaria en el sector farmacéutico. En este sentido, *VeChain* se ha asociado con varias empresas farmacéuticas para integrar identificadores únicos (ej. códigos QR/NFC en cada caja) en sus productos. Esto permite rastrear su recorrido y verificar su autenticidad a través de su *blockchain* pública (DrugPatentWatch, 2024). Un

³³ *Tamper-proof* hace referencia a sistemas o mecanismos diseñados para ser resistentes a manipulaciones no autorizadas, garantizando que los datos no puedan ser alterados sin dejar evidencia. En el contexto de *blockchain*, implica que, una vez registrada una información, no puede modificarse sin romper la integridad de toda la cadena.

paciente o farmacia puede escanear el código y consultar la procedencia del producto, asegurando que proviene de un fabricante legítimo y que no ha sido alterado. De forma similar, la empresa *Chronicled* en EE. UU. ha liderado el consorcio *MediLedger*, donde múltiples farmacéuticas y distribuidores comparten un *ledger permisionado* para verificar las transferencias de propiedad de medicamentos, cumpliendo con la DSCSA³⁴ (*Drug Supply Chain Security Act*). *Chronicled* asigna identidades digitales únicas a cada producto y registra cada movimiento en la *blockchain*, facilitando auditorías y previniendo la entrada de copias ilegítimas (DrugPatentWatch, 2024). Este sistema de serialización impide que, incluso si aparece un envase falsificado, su identidad digital concuerde con los registros de *blockchain*.

- **Plataformas consorciadas:** grandes proveedores tecnológicos han desarrollado soluciones de trazabilidad que integran IoT y *blockchain*. IBM adaptó su *blockchain Platform* basada en *Hyperledger Fabric* para el seguimiento de medicamentos, logrando visibilidad extremo a extremo en la cadena de suministro (DrugPatentWatch, 2024). Los diversos actores (fabricantes, mayoristas, farmacias, reguladores) operan nodos en una red privada y cada evento (producción, envío, recepción, dispensación) se registra de forma inmutable. Se pueden adjuntar datos IoT (por ejemplo, lecturas de sensores de humedad o geolocalización de vehículos) a las transacciones de *blockchain*. Estas soluciones han permitido mejorar la respuesta ante problemas como, por ejemplo, acelerar el retiro de lotes defectuosos.
- En el contexto de **industria farmacéutica 4.0**, la convergencia IoT-*blockchain* promete cadenas de suministro más resilientes y transparentes. Estudios durante la pandemia de COVID-19 resaltaron que la unión de estas tecnologías puede aumentar la flexibilidad y visibilidad de las operaciones y contribuir a su resiliencia (Chen et al., 2023). Uno de los modelos propuestos, integraba IoT y *blockchain* en hospitales para monitorizar stocks de medicamentos en tiempo real, detectando escasez o retrasos para responder ágilmente. Proyectos como el europeo *PharmaLedger*, exploraron el uso de *blockchain* en salud, incluyendo un piloto para incorporar datos de dispositivos

³⁴ La DSCSA o *Drug Supply Chain Security Act* es una ley de EE. UU. que establece un sistema nacional para rastrear y verificar medicamentos a lo largo de la cadena de suministro farmacéutica con el objetivo de mejorar la seguridad del paciente y combatir la falsificación.

IoT *wearables*³⁵ de pacientes en ensayos clínicos (Innovative Health Initiative, 2025).

Esto demuestra que sus aplicaciones no se limitan a la logística, sino que pueden abarcar cualquier flujo de datos donde la trazabilidad y la confianza sean esenciales.

2.2.9.4. Redes blockchain-IoT y desarrollo en Python

El ecosistema *Python* se ha vuelto especialmente relevante en los entornos IoT por su sencillez y amplio soporte para placas como *Raspberry Pi*. Afortunadamente, muchas soluciones *blockchain* para IoT, ya ofrecen una API³⁶ (*Application Programming Interface*) o un SDK³⁷ (*Software Development Kit*) en *Python*, facilitando su adopción en prototipos y desarrollos rápidos. IOTA cuenta con el cliente PyOTA³⁸ para el envío de transacciones o mensajes a la *Tangle*. Ethereum y cadenas EVM como *IoTeX* o *VeChain* se pueden gestionar usando librerías como *Web3.py*. También *Hyperledger Fabric* ofrece un SDK *Python* para enviar invocaciones a la red. Un punto interesante es la adopción de *MicroPython* o *CircuitPython* en microcontroladores IoT como el ESP32 (*Espressif Systems 32-bit Microcontroller*) o el STM32 (*STMicroelectronics 32-bit Microcontroller*). Si bien estas versiones de *Python* no pueden manejar directamente un cliente pesado de *blockchain*, sí permiten firmar datos localmente y comunicarse con gateways. Un sensor con *MicroPython* podría firmar sus lecturas con una clave privada y enviar el hash a un nodo *blockchain* externo para su registro. Adicionalmente, frameworks de *blockchain* ligeros como *EasyChain* o IOTA, podrían directamente correr en dispositivos más potentes como *Raspberry Pi*, permitiendo prototipos donde el nodo esté embebido localmente.

³⁵ Los *Wearables* son dispositivos electrónicos que se llevan puestos sobre el cuerpo, como relojes inteligentes, pulseras de actividad o sensores médicos. Permiten recopilar, procesar y transmitir datos en tiempo real, habitualmente en el contexto de salud, deporte o IoT.

³⁶ API o *Application Programming Interface* es un conjunto de definiciones y protocolos que permite que diferentes aplicaciones se comuniquen entre sí, facilitando el acceso a funcionalidades o datos de un sistema sin necesidad de conocer su implementación interna.

³⁷ SDK o *Software Development Kit* es un conjunto de herramientas, bibliotecas, documentación y utilidades que facilitan el desarrollo de aplicaciones sobre una plataforma específica, permitiendo a los desarrolladores integrar funcionalidades, acceder a APIs o compilar software de forma eficiente.

³⁸ Ahora obsoleto.

2.2.10. Desafíos

A pesar de los avances en el almacenamiento distribuido, *blockchain* e IoT, el estado del arte plantea desafíos técnicos que aún no se han resuelto completamente, tal y como queda reflejado en la **Tabla 5**. Creemos que estos retos justifican la necesidad de desarrollar nuevas soluciones que ahonden en la mejora de los sistemas de almacenamiento distribuidos.

Tabla 5. Resumen de los principales desafíos a los que se enfrenta el proyecto

Desafío	Descripción
Integridad de datos off-chain	Garantizar que los datos almacenados fuera de la <i>blockchain</i> (<i>off-chain</i>) no hayan sido alterados.
Sincronización entre nodos	Asegurar que todos los nodos mantengan una estructura de datos coherente a pesar de los posibles retardos, desconexiones o fallos de comunicación.
Escalabilidad	Superar las limitaciones de rendimiento y latencia, especialmente en sistemas con alta frecuencia de transacciones.
Rendimiento y consumo energético	Reducir el rendimiento y el impacto energético de los algoritmos de consenso sin comprometer la seguridad y la descentralización.
Gestión de identidad	Desarrollar mecanismos descentralizados para la autenticación y control de acceso sin depender de entidades centrales.
Nodos poco confiables	Asegurar la resiliencia del sistema ante nodos intermitentes o con recursos limitados, garantizando la disponibilidad y la replicación de datos.
Privacidad	Proteger los datos almacenados y transmitidos mediante cifrado, control de acceso y anonimización.

Fuente: elaboración propia.

2.2.11. Conclusión de estado del arte

El estado del arte actual muestra un panorama interesante, donde IoT y *blockchain* convergen para resolver problemas de confianza en entornos distribuidos. Ya existen soluciones enfocadas a la trazabilidad que aprovechan *blockchain* para asegurar los datos de sensores, con casos de éxito como los pilotos de *Modum* o *VeChain*. Por otro lado, hay soluciones orientadas al almacenamiento distribuido de datos (*IPFS*, *Sotri*, *Sia*, *Filecoin* o *Iagon*) que, en muchos casos, incorporan mecanismos de incentivos o verificación basados en *blockchain*.

No obstante, tras el análisis llevado a cabo (la **Tabla 6** muestra una comparativa de las distintas soluciones analizadas), no se han identificado soluciones de almacenamiento distribuido que combinen el conjunto de requisitos presentado al inicio de este proyecto:

1. Seguridad, trazabilidad y descentralización mediante tecnologías *blockchain*.
2. Diseñadas para ser ejecutadas en arquitecturas IoT de bajo consumo.
3. Que sean de código abierto disponible públicamente.

Tabla 6. Comparativa de las soluciones de almacenamiento analizadas

Solución	Open Source	Blockchain	Seguridad	Orientada IoT
IPFS	Si	No	No cifrado, no trazabilidad	Parcialmente ³⁹
STORJ	No ⁴⁰	Si	Si	No
SIA	Si	Si	Parcialmente ⁴¹	No
FILECOIN	Si	Si	Parcialmente ⁴²	No
IAGON	Si	Si	Si	No

Fuente: elaboración propia.

Creemos que dicha ausencia, justifica la necesidad de explorar nuevas propuestas que integren estas características de forma segura, eficiente, amigable y adaptable.

³⁹ IPFS no está específicamente diseñado para IoT, pero su arquitectura descentralizada y ligera lo hace compatible con entornos IoT, especialmente cuando se usa junto a gateways o nodos intermedios.

⁴⁰ Aunque el software de nodos y herramientas cliente es de código abierto, algunos componentes clave como los satélites (encargados de coordinar metadatos y reputación) son gestionados de forma centralizada por *Storj Labs* y no están completamente abiertos.

⁴¹ *Sia* cifra automáticamente todos los archivos antes de almacenarlos y registra en su *blockchain* los contratos entre usuarios y nodos, proporcionando seguridad y evidencia de almacenamiento, aunque no hay trazabilidad completa del ciclo de vida de un archivo (quién lo descargó, cuándo o con qué propósito).

⁴² *Filecoin* no cifra los datos por defecto, pero sí ofrece trazabilidad a través de su *blockchain*, donde se registran los contratos de almacenamiento, pruebas de retención y transferencias económicas entre usuarios y proveedores.

3. Objetivos y metodología de trabajo

A continuación, se presentan los objetivos que han guiado el desarrollo del proyecto y la estrategia adoptada para alcanzarlos.

3.1. Objetivo general

Diseñar e implementar un sistema de ficheros distribuido basado en tecnologías *blockchain* y P2P que permita al usuario almacenar, compartir y gestionar archivos, a través de dispositivos IoT de bajo consumo y coste, garantizando la trazabilidad, disponibilidad y soberanía de la identidad y del dato.

3.2. Objetivos específicos

A continuación, un listado con los objetivos específicos:

- **Analizar las limitaciones** de los sistemas tradicionales de almacenamiento centralizado y las oportunidades que ofrecen las tecnologías descentralizadas en la actualidad.
- **Investigar y seleccionar las tecnologías adecuadas** para la construcción de un sistema distribuido: dispositivos IoT, redes P2P, *blockchain*, mecanismos de cifrado, gestión de identidades y arquitecturas de microservicios.
- **Diseñar la arquitectura del sistema**, incluyendo sus componentes funcionales, la estructura de comunicación entre nodos, la capa de almacenamiento local, el modelo de metadatos y el uso de hashes para la identificación e integridad de los archivos.
- **Implementar un prototipo funcional** sobre dispositivos de bajo consumo (*Orange Pi One*), integrando los módulos principales: API REST, P2P, gestión de ficheros, gestión de identidades, cifrado y capa de consenso.
- **Evaluar el comportamiento del sistema** en escenarios de prueba, midiendo aspectos como la disponibilidad, la redundancia, la integridad de los datos, la recuperación ante fallos y la eficiencia del acceso distribuido.

- **Validar la capacidad del sistema para registrar operaciones** de forma trazable e immutable mediante *blockchain*, simulando casos de uso reales orientados a entornos industriales o de investigación.
- **Documentar el sistema**, su arquitectura y su implementación; liberar el código como proyecto de software libre con licencia *copyleft*.

3.3. Metodología de trabajo

El desarrollo del proyecto se estructura en varias fases que se abordan de forma iterativa e incremental, permitiendo revisar y ajustar decisiones técnicas a medida que avanza el trabajo.

La **Tabla 7** presenta un cronograma con la planificación:

1. **Fase de análisis:** revisión del estado del arte, identificación de tecnologías clave, análisis de requisitos funcionales y no funcionales. Se sientan las bases conceptuales y se determina la viabilidad técnica del sistema propuesto.
2. **Fase de diseño:** definición de la arquitectura del sistema, tanto a alto nivel (interacción entre componentes) como a nivel detallado (estructura de datos, flujos de comunicación, seguridad y redundancia).
3. **Fase de desarrollo:** implementación del prototipo utilizando lenguajes y herramientas adecuados para entornos IoT (ej. *Python*), configuración de los nodos y despliegue de microservicios. Integración de la capa P2P, API REST y sistema de almacenamiento basado en hash.
4. **Fase de integración *blockchain*:** selección de una plataforma adecuada (*IOTA*, *Tendermint*, *Cardano*, *Hyperledger*, etc.), implementación de los contratos inteligentes o lógica de consenso necesaria para la trazabilidad de operaciones y conexión con el sistema base.
5. **Fase de validación:** ejecución de pruebas funcionales y no funcionales (especialmente de seguridad y estrés) en distintos escenarios (fallo de nodos, concurrencia, latencias variables, etc.), recolección de métricas para la evaluación del rendimiento, la disponibilidad, la integridad de los datos y la resiliencia del sistema.

- 6. Fase de documentación y difusión:** redacción de esta memoria, publicación del código en un repositorio abierto, revisión de los resultados obtenidos y propuestas de mejora.

Tabla 7. Planificación temporal del trabajo

Fase	Tareas principales	Duración
1. Análisis del contexto	Revisión del estado del arte, definición de requisitos, análisis técnico	2 semanas
2. Diseño del sistema	Arquitectura general, diseño de componentes, modelo de datos, diseño de seguridad	3 semanas
3. Desarrollo del prototipo	Programación de microservicios, red P2P, almacenamiento distribuido y cifrado	4 semanas
4. Integración de blockchain	Selección de tecnología, integración de capa de consenso y trazabilidad	3 semanas
5. Pruebas y validación	Escenarios de fallo, métricas de rendimiento, redundancia, disponibilidad y seguridad	3 semanas
6. Documentación y conclusiones	Finalizar memoria con documentación técnica, conclusiones y líneas futuras	3 semanas
7. Revisión y entrega final	Revisión integral, ajuste de formato, correcciones y presentación	2 semanas

Fuente: elaboración propia.

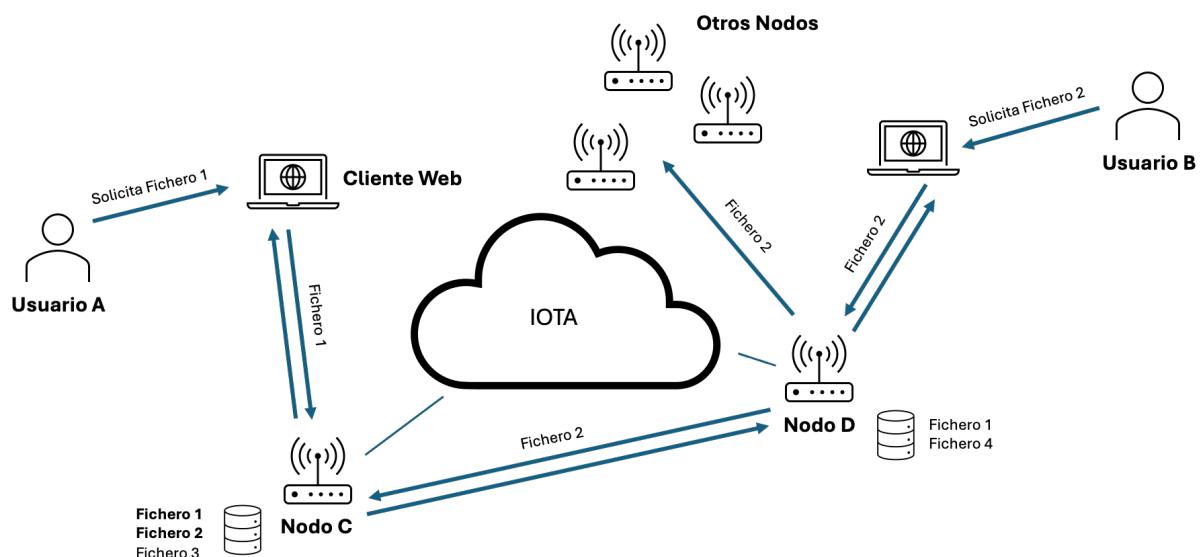
4. Análisis de requisitos

En este apartado se definen las necesidades y restricciones que han guiado el diseño e implementación del prototipo.

4.1. Descripción general

El sistema propuesto (al que hemos denominado *dfs3*), es una solución de almacenamiento distribuido de ficheros con cifrado. Diseñada específicamente para funcionar en redes de dispositivos IoT de bajo consumo como *Raspberry Pi*. Su finalidad es permitir a los usuarios almacenar, compartir y acceder a ficheros de forma segura, sin depender de infraestructuras centralizadas y aprovechando principios clave de la Web 3.0 como son la descentralización y la soberanía del dato o de la identidad digital.

Figura 12. Sistema de compartición de ficheros *dfs3*



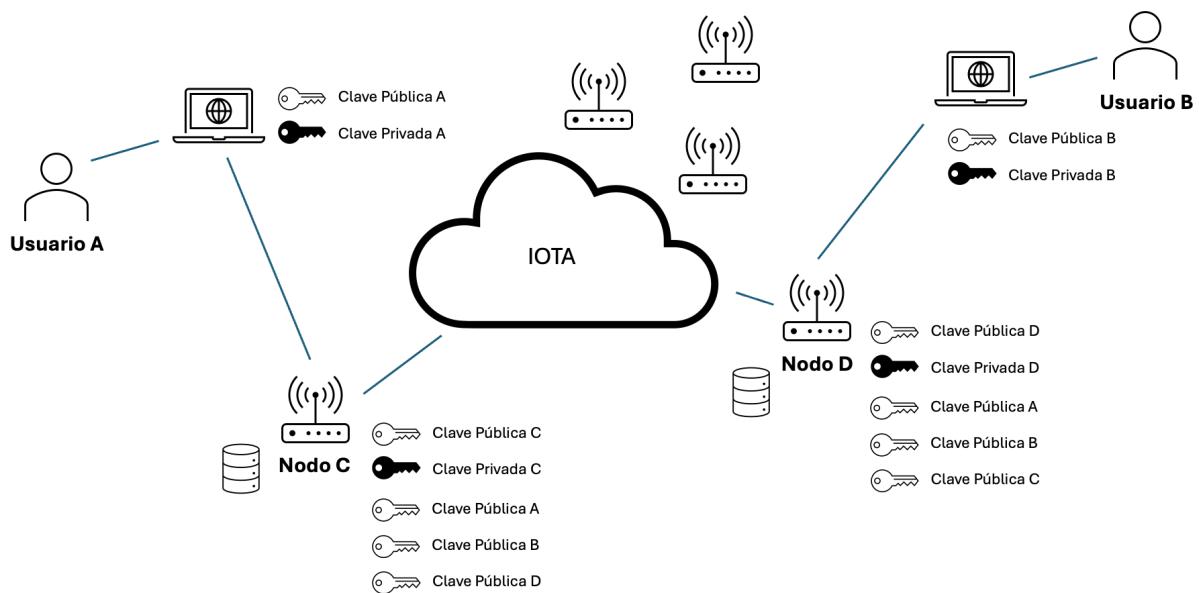
Fuente: elaboración propia.

La **Figura 12** ilustra el funcionamiento básico de la solución: cuando un usuario quiere acceder a un fichero, lo hace a través de su navegador web, conectando con su propio nodo (idealmente). Si este nodo almacena una copia del fichero, lo devuelve (ej. Fichero_1). En caso contrario (ej. Fichero_2) consulta sus metadatos y contacta con los nodos que lo sirvan,

devolviendo una copia al usuario y almacenándolo localmente para futuros accesos a modo de caché.

Tanto usuarios como nodos disponen de claves asimétricas que se utilizan para autenticarse y/o firmar (nodos y usuarios) o para cifrar (solo usuarios) los ficheros de su propiedad, tal y como queda reflejado en la **Figura 13**. La autenticación criptográfica basada en pares de claves garantiza que la identidad del usuario no dependa de un servidor central y le permite preservar su privacidad en un entorno abierto.

Figura 13. Identidad digital dfs3 basada en pares de claves



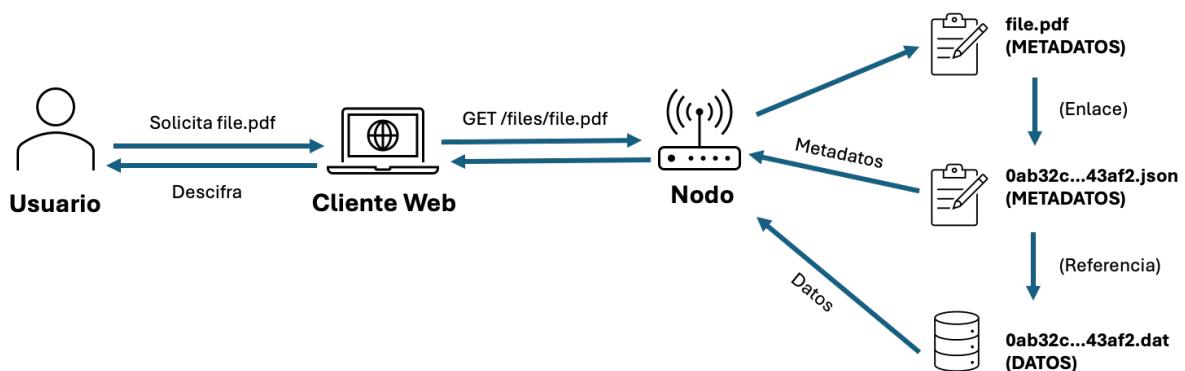
Fuente: elaboración propia.

Los nodos actúan como servidores ligeros. En principio, son accesibles para cualquier usuario registrado y exponen su API REST. Cada nodo gestiona su propio almacenamiento local, basado en una base de datos SQLite ligera y en una estructura de ficheros con entradas virtuales que básicamente son enlaces al fichero de metadatos con información sobre el fichero real. Estas entradas virtuales, actúan como una interfaz para el usuario autorizado. Le ofrecen una vista de su contenido y permiten gestionar la compartición de ficheros sin duplicar datos.

La **Figura 14** ilustra el acceso a un fichero. Cuando el usuario solicita un fichero (ej. file.pdf), el navegador lanza una petición API Rest a su nodo backend, que comprueba la existencia del

fichero en su *filesystem*. Dicho fichero, será una entrada virtual, apuntando al fichero de metadatos, cuyo nombre se identifica por el valor SHA-256 de su contenido con extensión JSON (ej. 0ab32c..43af2.json). El nodo devuelve los metadatos y los datos cifrados del fichero, almacenados en un archivo también identificado por su hash SHA-256, pero con extensión DAT (ej. 0ab32c..43af2.dat). Con los datos cifrados, los metadatos y el par de claves del usuario, el navegador descifra su contenido y lo presenta al usuario para su descarga o visualización.

Figura 14. Acceso al sistema de ficheros de dfs3



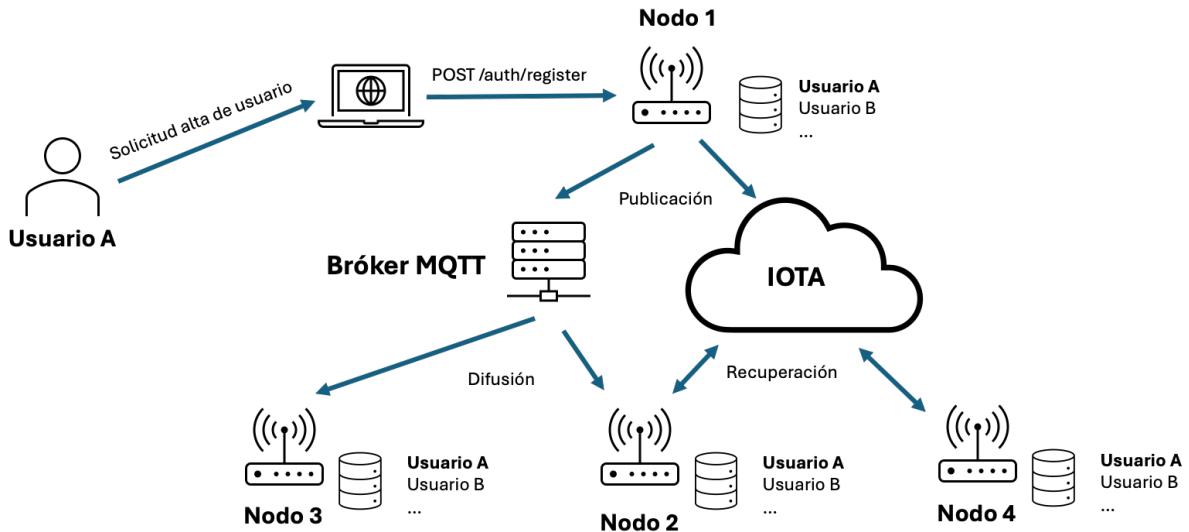
Fuente: elaboración propia.

Se implementa un modelo de replicación basado en eventos, en el que los ficheros pueden estar disponibles en múltiples nodos según su uso y compartición. La comunicación entre nodos se realiza a través de un sistema híbrido basado en:

- **Eventos publicados en la red IOTA**, que proporcionan un registro inmutable y descentralizado para la sincronización entre nodos y trazabilidad.
- **Mensajes MQTT de notificación**, utilizados como canal de control ligero para notificar al resto de nodos de la existencia de nuevos eventos en IOTA.

La **Figura 15** muestra un ejemplo de replicación y sincronización por eventos: cuando un usuario solicita su alta en el sistema, su petición se almacena en la red IOTA como evento JSON. Se informa a cada nodo mediante un mensaje MQTT. Cada nodo, conecta con IOTA, descarga una copia del mensaje, verifica su firma y actúa según el tipo de evento.

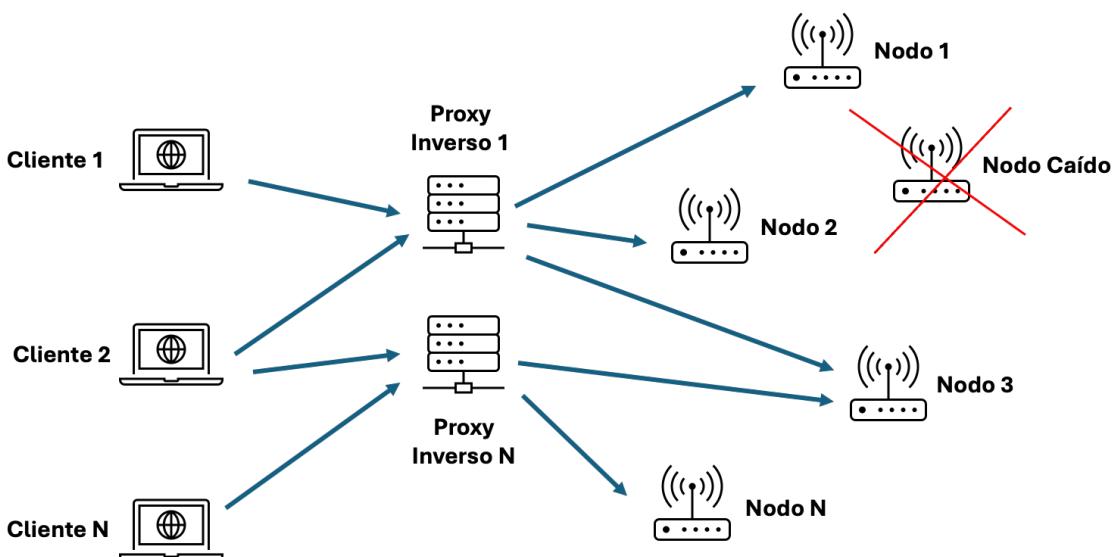
Figura 15. Sistema de replicación por eventos y publicación en IOTA dfs3



Fuente: elaboración propia.

Un punto interesante a tener en cuenta es que, el usuario puede interactuar a través de cualquier nodo disponible sin necesidad de estar vinculado a un servidor. Esto permite tolerancia a fallos, movilidad y descentralización. Los ficheros se cifran en origen (navegador), mediante claves simétricas, antes de ser enviados para garantizar la confidencialidad en todo momento, incluso en caso de que un nodo pueda verse comprometido.

Figura 16. Propuesta de escalabilidad horizontal y resiliencia en dfs3



Fuente: elaboración propia.

Una de las fortalezas de este sistema es que, gracias a su arquitectura descentralizada, cualquier usuario puede acceder a su información desde cualquier nodo disponible en la red. Por ejemplo, al combinar esta característica con un sistema de proxy inverso con balanceo como el mostrado en la **Figura 16**, se obtiene una solución que escala horizontalmente y es tolerante a fallos, apta para dar servicio a múltiples usuarios, incluso ante la caída de nodos.

4.2. Requisitos funcionales

En la **Tabla 8** se enumera la lista de requisitos funcionales definidos dentro del alcance:

Tabla 8. Listado de requisitos funcionales definidos para dfs3

Código	Nombre	Descripción breve
RF01	Alta de usuario	Permitir a un usuario registrarse en cualquier nodo, generar claves y propagarse en la red.
RF02	Autenticación de usuario	Validar al usuario mediante desafío firmado con clave privada en cualquier nodo.
RF03	Alta de nodo	Permitir a un nodo registrarse y publicar su identidad en la red.
RF04	Actualización de estado de nodo	Enviar periódicamente información de <i>uptime</i> , IP, puerto y versión al resto de nodos.
RF05	Subida de ficheros	Permitir al usuario la subida de ficheros (cifrados), registrando una entrada virtual para futuros accesos.
RF06	Descarga de ficheros	Permitir al usuario la descarga de ficheros desde cualquier nodo con descifrado local basado en su identidad (clave privada).
RF07	Compartición de fichero	Permitir al propietario, desde cualquier nodo, compartir ficheros cifrados con otros usuarios de forma segura.
RF08	Listado de ficheros	Permitir al usuario listar todos los ficheros en su espacio virtual desde cualquier nodo.
RF09	Renombrado de fichero virtual	Permitir al usuario cambiar el nombre de un fichero (entrada virtual) desde cualquier nodo.
RF10	Borrado de fichero	Permitir al usuario eliminar sus entradas virtuales y, si aplica, el fichero desde cualquier nodo.
RF11	Replicación de ficheros en nodos	Permitir a cualquier nodo la replicación de ficheros para asegurar la redundancia en la red.
RF12	Propagación de eventos	Publicar en IOTA y MQTT todos los cambios relevantes, por trazabilidad y para sincronización entre nodos.

Fuente: elaboración propia.

4.3. Requisitos no funcionales

La **Tabla 9** lista los requisitos no funcionales que establecen los criterios de calidad, rendimiento, seguridad y usabilidad del sistema.

Tabla 9. Listado de requisitos no funcionales

Código	Nombre	Descripción breve
RNF01	Seguridad de datos	Todos los ficheros deben almacenarse cifrados, protegiendo la confidencialidad en reposo.
RNF02	Descentralización	El sistema debe operar sin servidores centrales, usando nodos autónomos de bajo coste.
RNF03	Alta disponibilidad	El sistema debe tolerar la caída de nodos sin perder el acceso a los datos.
RNF04	Integridad de la información	Todos los eventos deben estar firmados y ser verificables desde cualquier nodo.
RNF05	Escalabilidad	El sistema debe permitir añadir nuevos nodos y usuarios sin pérdida de rendimiento, hasta un número de elementos razonable (ej. 100 nodos).
RNF06	Consistencia de visibilidad	Todos los nodos deben mantener una estructura coherente del sistema de ficheros distribuido.
RNF07	Optimización del espacio	No debe duplicarse contenido entre nodos innecesariamente.
RNF08	Interoperabilidad	El cliente web debe ser compatible con los principales navegadores actuales.
RNF09	Confidencialidad de claves	La clave privada de usuario no debe salir del dispositivo y estará protegida por una <i>passphrase</i> .
RNF10	Tamaño máximo de ficheros permitido	Se debe limitar a un máximo permitido el tamaño de ficheros (ej. 100 MB).
RNF11	Control de tipos de fichero	Se debe validar el tipo MIME y, por seguridad, bloquear los tipos no permitidos (ej. ejecutables).
RFN12	Control de sesión por inactividad	Para reforzar la seguridad, el sistema debe cerrar automáticamente la sesión del usuario tras un tiempo de inactividad establecido (ej. 5 minutos).

Fuente: elaboración propia.

4.4. Restricciones y limitaciones

El proyecto se ha desarrollado bajo un conjunto de condiciones (técnicas y de alcance) que han condicionado las decisiones de diseño. En la **Tabla 10**, se presentan tanto las restricciones

impuestas por el propio entorno de ejecución, como las asumidas voluntariamente para acotar la complejidad y asegurar la viabilidad en el plazo de tiempo marcado.

Tabla 10. Principales limitaciones y restricciones del proyecto

Categoría	Descripción
Hardware	El sistema debe ejecutarse en dispositivos de bajo consumo, limitando CPU, RAM y almacenamiento disponibles.
Red	Se asume una red potencialmente inestable, con posibles latencias elevadas, bajo ancho de banda y desconexiones entre nodos.
Persistencia de eventos	La propagación de eventos depende de la red IOTA; errores o caídas en la Tangle pueden afectar a la sincronización.
Modelo de consistencia	Se adopta un modelo de consistencia eventual ⁴³ en lugar de consistencia estricta para simplificar la replicación.
Estructura de archivos	El sistema no soporta directorios anidados; todos los ficheros se almacenan en un espacio virtual raíz por usuario.
Escalabilidad	El diseño está orientado a pequeños entornos domésticos o de investigación, no a despliegues de alta escala o de uso intensivo.
Gestión de versiones	Cada modificación de un archivo genera un nuevo hash y objeto; no se implementa control de versiones.
Gestión de permisos simplificada	Solo se contempla el permiso de acceso. No hay roles, jerarquías ni permisos granulares por carpetas o grupos.
Autenticación por sesión	Para simplificar el control de sesión, éste se ha basado en tokens locales, en lugar de estándares distribuidos como JWT.
Gestión local de claves	Las claves privadas de usuario se almacenan en el navegador (protegidas con <i>passphrase</i>), sin mecanismos de recuperación o de rotación.
Replicación estática	Cada fichero se replica a M nodos (por defecto 2 además del nodo propietario), elegidos determinísticamente, sin redistribución dinámica por cambio de red.
Sin recolección de basura automática	No se implementan mecanismos para eliminar réplicas obsoletas o gestionar espacio ocupado por los usuarios eliminados.
Replicación sin erasure coding	El sistema se basa en copias completas de archivos, lo que implica un mayor consumo de espacio y una menor eficiencia de almacenamiento, en comparación con esquemas de codificación como <i>Reed-Solomon</i> .
No hay recuperación de cuentas	Si un usuario pierde su clave privada o <i>passphrase</i> , no hay forma de recuperar el acceso a sus ficheros.

⁴³ La consistencia eventual es un modelo en sistemas distribuidos según el cual, aunque los datos no estén sincronizados en todos los nodos en un momento dado, todos acabarán alcanzando un estado coherente tras un periodo de tiempo, siempre que no se produzcan nuevas actualizaciones.

Categoría	Descripción
No se contempla control de cuota	No se ha contemplado un sistema de cuotas por usuario.
Sin detección automática de conflictos	No hay detección de inconsistencias entre nodos más allá de la replicación controlada y determinista.
Edición de datos no implementada	Los usuarios no pueden modificar su nombre, alias, tags, ni gestionar sus claves una vez registrados.
Baja de usuarios y nodos no disponible	No existe la funcionalidad para eliminar lógicamente a un usuario o nodo.
Sin UI para monitorización / administración	No hay interfaz para gestión avanzada del sistema por parte de administradores y/o supervisores.

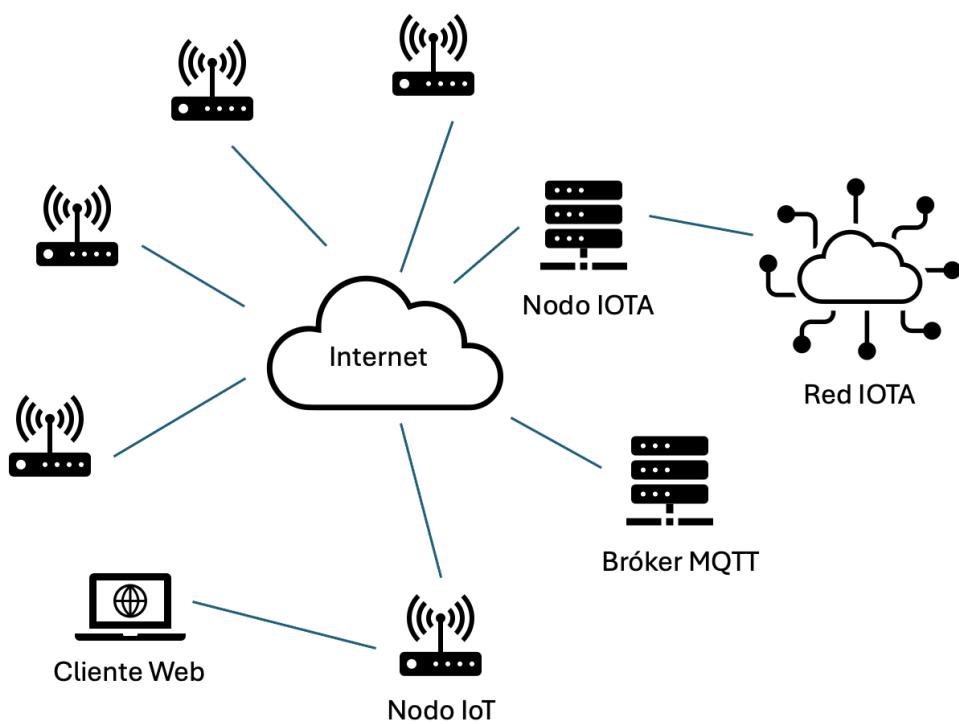
Fuente: elaboración propia.

Estas restricciones y limitaciones han sido consideradas durante el diseño para ayudar a focalizar el esfuerzo en los objetivos principales y facilitar una implementación funcional realista dentro del alcance definido.

5. Diseño del sistema

El diseño está orientado a maximizar la resiliencia, la simplicidad operativa y la protección de los datos. Utiliza tecnologías como IOTA y MQTT para la sincronización de eventos y mecanismos de cifrado de extremo a extremo para garantizar la confidencialidad de la información. La solución apuesta por una arquitectura modular que facilite las futuras extensiones y/o adaptaciones de nuevos casos de uso.

Figura 17. Arquitectura general de la solución



Fuente: elaboración propia.

5.1. Arquitectura general

El diseño del sistema sigue una arquitectura distribuida y descentralizada basada en nodos ligeros, como muestra la **Figura 17**. Se compone de los siguientes elementos:

- **Cliente:** aplicación web que se ejecuta en el navegador. Puede cargarse desde el propio nodo o como contenido estático “cacheable” desde un servidor HTTPS alternativo. Gestiona la clave privada del usuario, almacenada localmente y protegida mediante

passphrase. La comunicación entre el cliente y el nodo es a través de API REST (HTTPS).

Requiere que el cliente se identifique previamente.

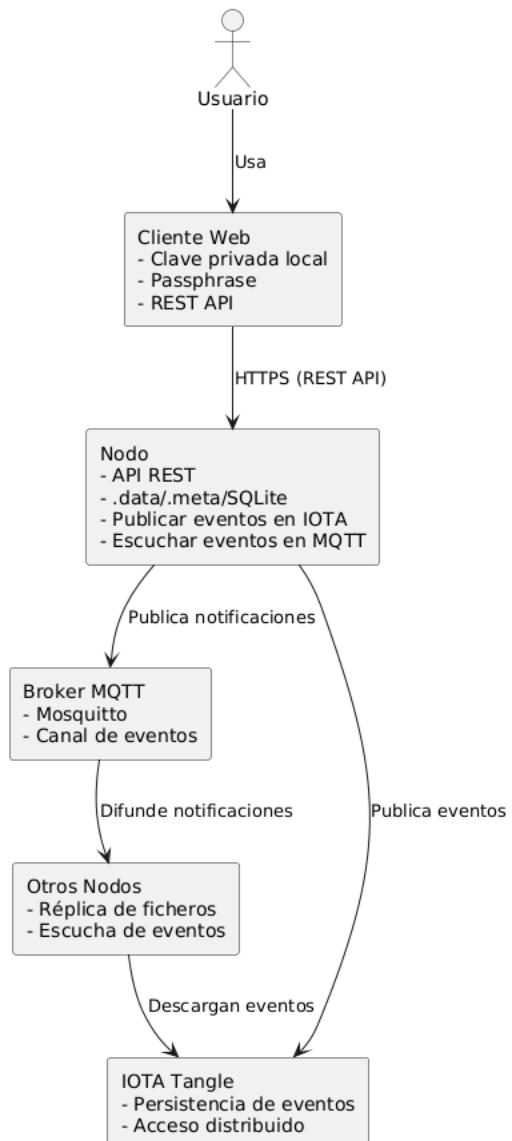
- **Nodo:** cada nodo funciona como un *backend* que expone su API REST para interactuar con el cliente web. Internamente, cada nodo almacena los ficheros cifrados, gestiona los metadatos, mantiene una base de datos ligera (SQLite) para la sincronización de información de usuarios, nodos e histórico de eventos, controla el acceso a los datos mediante un sistema de archivos basado en entradas virtuales y publica / gestiona los eventos para asegurar un registro inmutable y descentralizado.
- **Bróker MQTT:** un servicio de mensajería ligera, utilizado para distribuir las notificaciones sobre nuevos eventos. Permite a cada nodo reaccionar ante cambios en la red.
- **IOTA Tangle:** red *blockchain* que registra el detalle de cada evento (alta de usuario, creación de fichero, replicación, etc.). Los eventos se firman antes de su publicación en IOTA, permitiendo la validación independiente de cada nodo. Para las pruebas se ha desplegado un nodo propio (Hornet).
- **Otros Nodos:** resto de participantes en la red. Actúan como almacenamiento redundante, reciben notificaciones y eventos sobre los que actúan y actualizan su estado periódicamente.

5.1.1. Comunicación entre componentes

La **Figura 18** representa el flujo de interacción entre los distintos componentes, desde el cliente web hasta los nodos distribuidos:

- El cliente web (navegador) se comunica con su nodo a través de los diferentes *endpoints* de la API REST.
- Los nodos publican eventos completos en la red IOTA y avisan a través de MQTT.
- Los nodos no se comunican directamente entre sí para operaciones de control. Sincronizan su estado observando el flujo de eventos publicado en IOTA / MQTT.

Figura 18. Diagrama de la arquitectura de la solución



Fuente: elaboración propia.

5.2. Justificación de elección tecnológica

En ese apartado, se justifican las principales decisiones tecnológicas tomadas en el diseño y desarrollo de la solución:

5.2.1. IOTA Tangle

Se ha seleccionado IOTA como red descentralizada para el registro de eventos por varias razones:

- **Persistencia inmutable:** los eventos almacenados en la *Tangle* no pueden ser modificados ni eliminados posteriormente. Esto garantiza la integridad histórica de todo el sistema.
- **Escalabilidad sin comisiones:** a diferencia de otras *blockchains* tradicionales, IOTA no requiere de comisiones por transacción. Esto permite registrar eventos de manera gratuita y sostenible.
- **Modelo ligero:** IOTA está optimizado para entornos de IoT, con dispositivos de bajo consumo como los usados en este proyecto.

Aunque IOTA es una red descentralizada, como parte de la arquitectura se ha **desplegado un nodo IOTA propio**. Esta decisión permite interactuar con la *Tangle* de manera directa, sin depender de nodos públicos externos. El uso de un nodo IOTA propio aporta múltiples ventajas, como el control total sobre la publicación de eventos, la mejora de la latencia en la validación y la recuperación de eventos. **Evita la realización de PoW** en los nodos y simplifica en general el desarrollo y despliegue: no es necesario instalar el SDK de IOTA, cuyo bind para Python aún no está lo suficiente maduro, en cada nodo.

5.2.2. MQTT

Se ha adoptado MQTT como mecanismo de comunicación entre nodos por varias razones:

- **Bajo consumo de ancho de banda:** ideal para notificaciones breves y frecuentes como nuevos eventos.
- **Modelo publish / subscribe:** permite una arquitectura desacoplada y reactiva. Los nodos se comunican sin necesidad de conexiones directas entre sí.
- **Implementación sencilla:** la configuración y uso del servidor *Mosquitto* permite una implementación ligera, ideal para esta primera versión con pocos nodos.

MQTT es una tecnología extremadamente ligera y fácilmente escalable. Diseñada para redes con gran cantidad de dispositivos. Esta inclusión no condiciona la escalabilidad ni compromete la filosofía de descentralización del sistema.

Aunque el proyecto persigue el objetivo de una arquitectura distribuida y descentralizada, se ha incorporado un **servidor MQTT como mecanismo centralizado de notificación**. Esta decisión responde a necesidades prácticas de eficiencia en la propagación de eventos: permite que los nodos sean alertados cuando se genera un nuevo evento, sin necesidad de realizar costosas consultas periódicas a IOTA o tener que implementar complejos sistemas de descubrimiento distribuido como DHT. El servidor MQTT no almacena información crítica ni estados de la red, únicamente mensajes de control de bajo volumen (identificadores de eventos en IOTA). La persistencia e integridad de los datos se mantiene descentralizada a través de la *Tangle*. Además, se podría reemplazar fácilmente por mecanismos distribuidos alternativos en versiones futuras.

5.2.3. SQLite

Para la gestión de datos locales (usuarios, nodos y ficheros) se ha optado por una base de datos ligera SQLite por varias razones:

- **Base de datos embebida:** no requiere de un servidor separado y facilita la instalación en dispositivos con recursos limitados.
- **Fiabilidad y rendimiento:** adecuada para volúmenes de datos moderados y para operaciones transaccionales ligeras.
- **Compatibilidad universal:** puede integrarse fácilmente con aplicaciones Python y otros lenguajes de programación.

5.2.4. Cliente Web

El cliente se implementa como una aplicación web estática basada en HTML5 y JavaScript por los siguientes motivos:

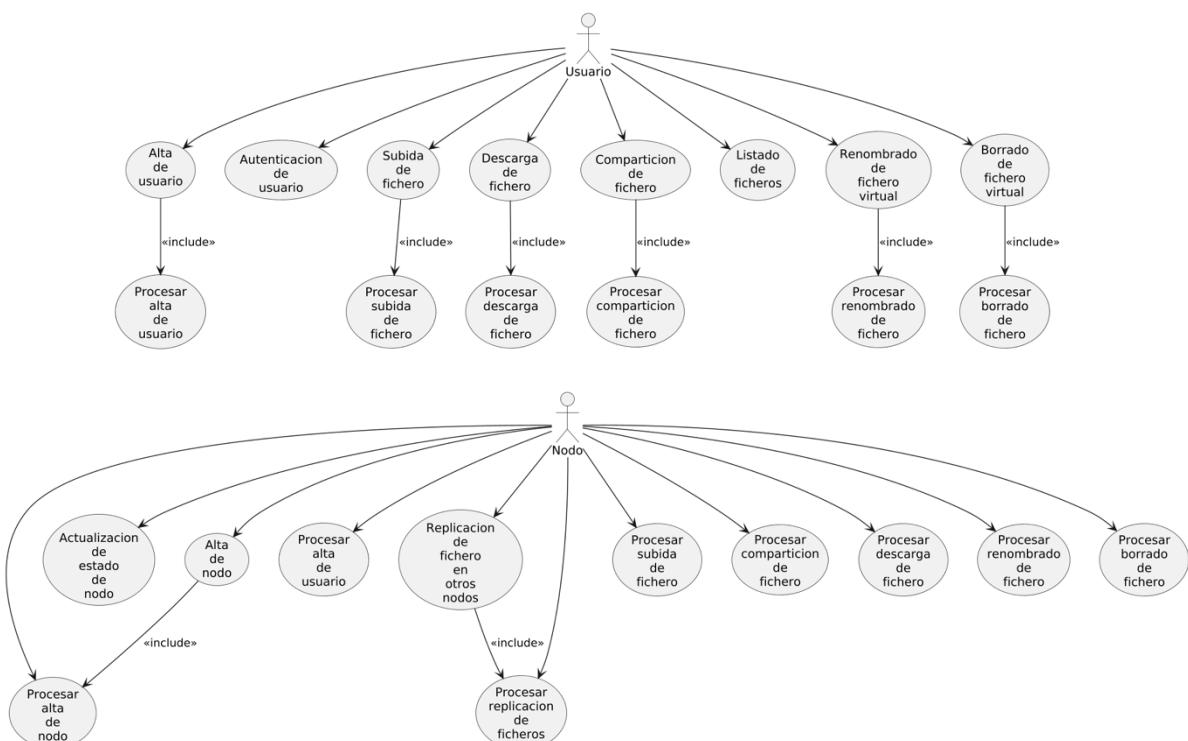
- **Accesibilidad:** cualquier dispositivo con un navegador relativamente moderno puede acceder al sistema sin necesidad de instalar software adicional.
- **Seguridad:** librerías como *WebCrypto API* permiten realizar operaciones criptográficas (generación de claves, firma, cifrado, descifrado) de manera segura dentro del propio navegador, manteniendo la clave privada bajo el control exclusivo del usuario.

- **Experiencia de usuario:** permite desarrollar interfaces amigables para una gestión de ficheros sencilla.

5.3. Modelo de casos de uso

La **Figura 19** muestra un diagrama con los principales casos de uso, proporcionando una visión general de las operaciones definidas. La **Tabla 11** describe la lista de casos de uso y acciones a realizar por parte del usuario.

Figura 19. Diagrama general de casos de uso



Fuente: elaboración propia.

Tabla 11. Listado de casos de uso

Código	Nombre del Caso de Uso	Breve Descripción
CU01	Alta de usuario	Permite a un usuario registrarse, generando su par de claves y publicando el alta para replicar al resto de nodos.
CU02	Autenticación de usuario	Permite al usuario autenticarse mediante la firma de un desafío generado por el nodo.
CU03	Alta de nodo	Un nuevo nodo genera su propia clave, publica su alta en IOTA y se lo notifica al resto para su sincronización.

Código	Nombre del Caso de Uso	Breve Descripción
CU04	Actualización de estado de nodo	Cada nodo envía periódicamente su estado actual (<i>uptime</i> , IP, puerto, etc.) a través de un evento en IOTA / MQTT.
CU05	Subida de fichero	El usuario sube un fichero al nodo. Se registra su entrada. Se publica su creación al resto de nodos para sincronización y por redundancia.
CU06	Descarga de fichero	El usuario solicita un fichero. El nodo entrega su contenido cifrado y la clave compartida (cifrada para ese usuario). El navegador descifra su contenido y el usuario a los datos.
CU07	Compartición de fichero	Un usuario autoriza a otro a acceder a un fichero de su propiedad. Comparte la clave con él. Crea una nueva entrada en su espacio virtual y difunde el cambio para sincronizar al resto de nodos.
CU08	Listado de ficheros	El usuario visualiza la lista de ficheros a los que tiene acceso, incluyendo propios y compartidos.
CU09	Renombrado de fichero	El usuario puede cambiar el nombre de un fichero (entrada). El cambio se propaga para mantener la coherencia entre nodos.
CU10	Borrado de fichero	El usuario puede eliminar la entrada de un fichero. Si no quedan referencias, también se borra su contenido. El evento se difunde a toda la red.
CU11	Replicación de fichero en otros nodos	Siguiendo un algoritmo determinista, los nodos pueden replicar un fichero para garantizar su redundancia. Se descarga una copia del contenido cifrado y se comunica al resto de nodos para actualizar su localización.

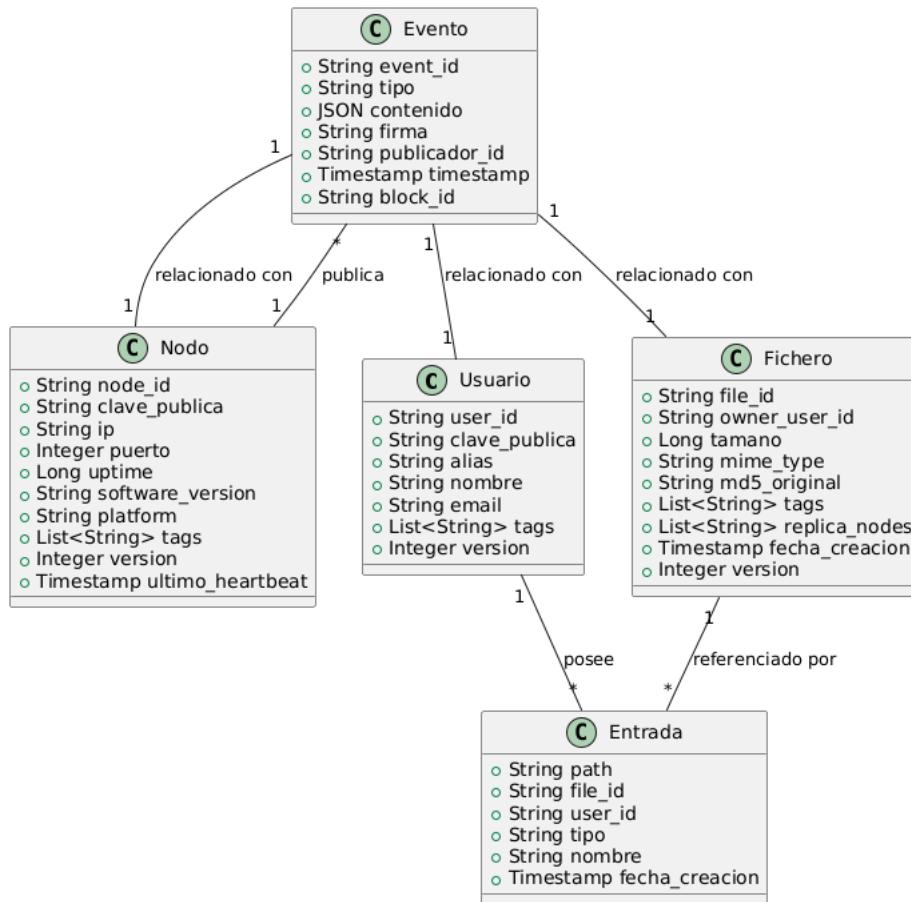
Fuente: elaboración propia.

Para una descripción detallada de los diferentes casos de uso definidos, se puede consultar el anexo [Casos de uso extendidos](#).

5.4. Modelo de clases

La **Figura 20** muestra el diagrama de clases UML, representando las asociaciones y dependencias. La **Tabla 12** incluye un listado de las entidades definidas durante la fase de diseño. Para una descripción detallada de cada clase, se puede consultar el anexo [Modelo de clases](#).

Figura 20. Diagrama de clases UML



Fuente: elaboración propia.

Tabla 12. Entidades del modelado de clases

Clase	Rol
Usuario	Representa a cada usuario registrado en el sistema. Cualquiera con acceso a un nodo puede crear un usuario sin necesidad de entidades centrales.
Nodo	Representa a cada nodo registrado en la red. Cualquiera puede crear y registrar un nodo nuevo si permite la comunicación desde/hacia Internet.
Fichero	Representa el contenido real cifrado, referenciado por su <code>file_id</code> que se genera a partir de su contenido.
Entrada Virtual	Representa la visibilidad del fichero en el espacio virtual del usuario (nombre, ruta). Apunta a una estructura de metadatos asociada al fichero de datos cifrado.
Evento	Representa cualquier suceso o evento publicado en IOTA (alta de usuario, alta de nodo, creación de fichero, etc.). Se comunica mediante mensajes MQTT.

Fuente: elaboración propia.

5.5. Modelo de datos

El diseño del modelo de datos define la estructura interna de la información almacenada. Se utilizan dos mecanismos de persistencia:

- Archivos en disco para el almacenamiento de los ficheros cifrados, metadatos y entradas virtuales de cada usuario.
- Una base de datos ligera (SQLite) para gestionar información estructurada sobre usuarios, nodos, eventos y relaciones entre sí.

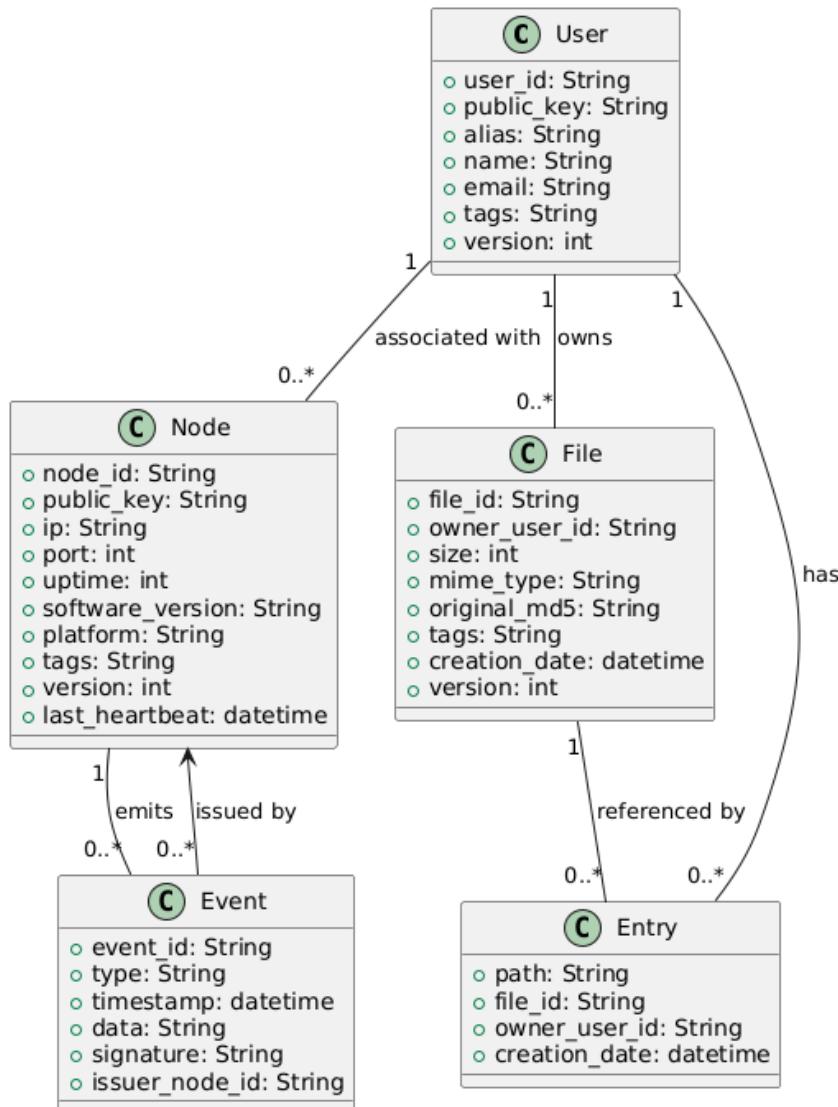
Tabla 13. Entidades del modelo de datos

Entidad	Descripción
Usuario	Identidad única (<code>user_id</code>) basada en un par de claves criptográficas. Se identifica de forma única a partir del hash SHA-256 de su clave pública. Cada usuario posee un alias que facilita su identificación.
Nodo	Representa a un servidor participante en la red. Identificado de forma única (<code>node_id</code>) a partir de su clave pública. Gestiona el almacenamiento local de los ficheros cifrados y la publicación de eventos.
Fichero	Representa el contenido almacenado. Se identifica mediante su hash SHA-256 (<code>file_id</code>) y lleva asociado un conjunto de metadatos (tamaño, propietario, fechas, usuarios autorizados, etiqueta, etc.).
Entrada Virtual	Representa una vista para el usuario del fichero. Se identifica por un nombre (<code>filename</code>) y está vinculado al fichero de metadatos. Permite tener múltiples vistas de un mismo contenido sin duplicarlo.
Evento	Registro de acciones como la creación o compartición de ficheros, actualización de nodos, etc. Se publica en IOTA / MQTT para garantizar un historial inmutable y descentralizado. Útil para auditoría y sincronización inicial de nodos.

Fuente: elaboración propia.

La **Tabla 13** describe las principales entidades del modelo de datos. La **Figura 21** ilustra gráficamente las relaciones entre estas entidades, destacando la separación entre almacenamiento físico (ficheros) y su representación lógica (entradas virtuales). Para un esquema completo de la base de datos y el código SQL de creación, se puede consultar el anexo “[Modelo de base de datos](#)”.

Figura 21. Diagrama entidad relación del modelo de datos



Fuente: elaboración propia.

5.6. Consideraciones de seguridad

La seguridad de la información ha supuesto un aspecto crítico en el diseño de la solución por su enfoque orientado a la protección de los datos dentro de un entorno distribuido no confiable. Para garantizar la confidencialidad, integridad y autenticidad de los datos y de las acciones llevadas a cabo por el usuario, se han adoptado diversas medidas específicas:

- **Protección de la clave privada de usuario:** cada usuario dispone de un par de claves criptográficas almacenadas en su navegador que garantizan su identidad y permiten el cifrado de ficheros extremo a extremo. La clave privada se genera y almacena

localmente en el navegador. Está protegida por un mecanismo de acceso seguro basado en *passphrase*.

- **Protección de la clave privada del nodo:** durante la fase de registro inicial, cada nodo genera un par de claves criptográficas para su identificación y firma de eventos. La clave privada se cifra localmente, empleando una clave derivada de la palabra de paso introducida por el operador. Cada vez que el nodo se inicia, es imprescindible proporcionar esta palabra de paso para descifrar y cargar en memoria la clave privada. Incluso si se compromete el nodo, la clave privada permanece inaccesible.
- **Cifrado en origen:** antes de enviar un fichero al nodo, se cifra en el navegador del cliente con una clave compartida aleatoria. A su vez, ésta se cifra con la clave pública del propietario y queda registrada en los metadatos del fichero. Este enfoque garantiza que los datos permanecen protegidos tanto en reposo como durante la transmisión, impidiendo el acceso a su contenido, aún en el caso de que el nodo pueda verse comprometido.
- **Verificación de la integridad de eventos mediante firma digital:** todos los eventos generados y publicados se firman digitalmente por el nodo emisor. Esto permite al nodo receptor verificar la autenticidad e integridad del evento. De este modo, se protege el sistema contra la manipulación o suplantación de identidad y se asegura la fiabilidad de la información propagada.

5.7. Consideraciones de sincronización y replicación

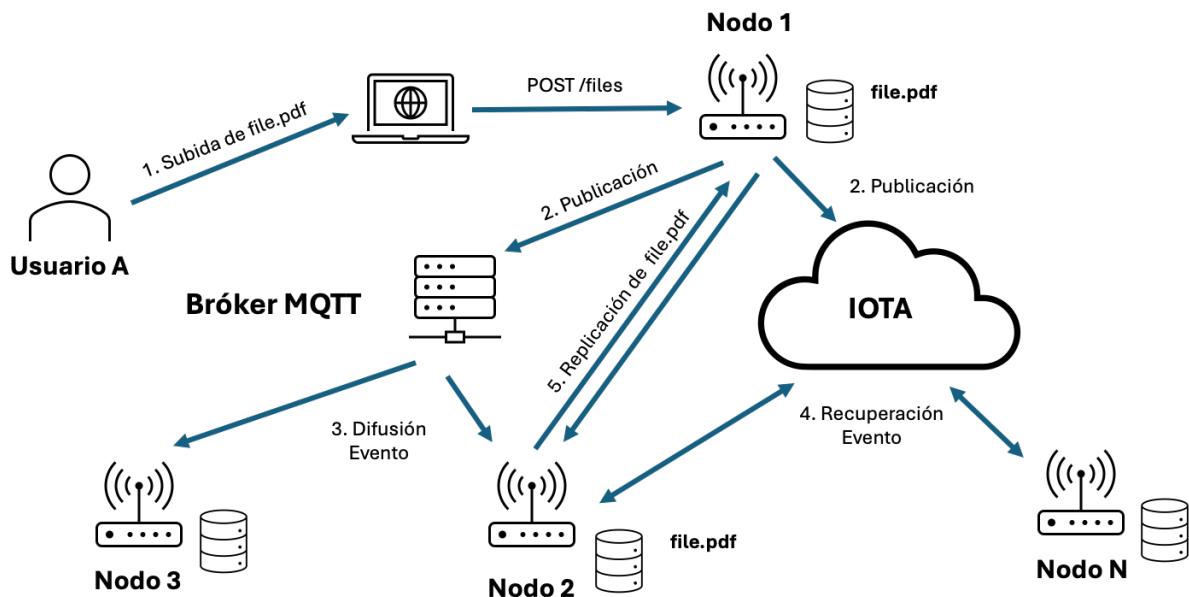
La sincronización y la replicación de los datos son componentes esenciales. A continuación, se describen las estrategias y mecanismos implementados:

- **Sincronización de entradas entre nodos:** la sincronización se basa en la propagación de eventos a través de la red IOTA / MQTT. Cada vez que se crea, modifica, elimina o comparte un fichero se genera un evento firmado que se publica en la *Tangle* y se comunica al resto de nodos a través del bróker MQTT. Todos los nodos suscritos recuperan el evento y actualizan localmente sus estructuras de acuerdo con la información recibida. De esta forma, se mantiene la coherencia de los espacios de usuario entre nodos sin necesidad de sincronización directa.

- **Estrategias de replicación para garantizar la disponibilidad:** el sistema implementa replicación de ficheros, basada en eventos de compartición y en políticas de disponibilidad / almacenamiento por nodo. Cuando un fichero se comparte con un usuario registrado en un nodo diferente, puede replicarlo localmente con tan solo acceder a él. De este modo, la carga de almacenamiento se distribuye progresivamente y se mejora la disponibilidad, incluso en escenarios de desconexión parcial o fallo de nodos individuales. Aunque inicialmente se fuerza una replicación activa básica para garantizar una disponibilidad mínima ($M=2$), el sistema propicia el crecimiento de réplicas a medida que los usuarios utilizan los ficheros compartidos / accedidos con más frecuencia. Se puede consultar el algoritmo de replicación implementado en el anexo *Lógica de replicación de ficheros*.
- **Gestión de coherencia eventual mediante eventos distribuidos:** se adopta un modelo de coherencia eventual, en el que no se requiere que todos los nodos reflejen simultáneamente el mismo estado. A través de la publicación y recuperación de eventos, los cambios se propagan progresivamente hasta que todos los nodos alcanzan un estado consistente. Este enfoque, combinado con el almacenamiento de eventos firmados e inmutables, proporciona robustez frente a la pérdida de mensajes por desconexión temporal a costa de aceptar períodos más o menos breves de inconsistencia transitoria.

La **Figura 22** ilustra el proceso completo de replicación. Cuando un usuario sube un fichero (ej. file.pdf) a su nodo (ej. nodo1), se almacena y se publica un evento en IOTA / MQTT. El resto de nodos reaccionan y, si se dan unas condiciones deterministas (ver anexo “*Lógica de replicación de ficheros*”), el nodo seleccionado (ej. nodo2) conecta con el que ha publicado el evento (ej. nodo1) para solicitar una copia de los datos. Una vez completado, publica un nuevo evento en IOTA / MQTT para que el resto de nodos actualice su lista de fuentes en los metadatos del fichero replicado.

Figura 22. Estrategia de replicación en dfs3



Fuente: elaboración propia.

La decisión de **replicar cada fichero en tres nodos** responde a un compromiso entre redundancia, eficiencia y simplicidad. Este número, permite tolerar la caída simultánea de hasta dos nodos sin pérdida de datos, lo que resulta adecuado en redes descentralizadas con recursos limitados. Tres réplicas favorecen la distribución de carga, permiten esquemas de quorum simples (como 2 de 3) y son práctica habitual en sistemas distribuidos consolidados como HDFS o Ceph. No obstante, el valor es configurable y puede ajustarse en función del tamaño de la red, la fiabilidad de los nodos y las necesidades de tolerancia a fallos. En lo que respecta al **tamaño máximo de fichero, limitado a 100MB**, se busca evitar la saturación de recursos en dispositivos IoT y garantizar una replicación eficiente. Este valor se considera un compromiso razonable entre rendimiento, fiabilidad y flexibilidad. De nuevo, puede ajustarse según las capacidades y el uso de la red previsto.

6. Implementación

A continuación, pasamos a describir el proceso de desarrollo e implementación del prototipo. Se detallan las tecnologías empleadas, organización del código y módulos definidos. Se aborda la lógica del *backend* y del cliente web (*frontend*), así como su integración con los servicios IOTA y MQTT.

6.1. Entorno de desarrollo

Comenzamos con una descripción del entorno de desarrollo, lenguajes de programación, herramientas y librerías empleados.

6.1.1. Tecnologías y herramientas

La elección de cada componente responde a criterios de rendimiento, compatibilidad con dispositivos IoT de bajo consumo, madurez tecnológica y adecuación al enfoque del proyecto:

- **Lenguajes de programación**
 - Python: lenguaje de programación interpretado, de alto nivel y propósito general. Se utiliza ampliamente en ciencia de datos y desarrollo de software, especialmente *backends*, aplicaciones web y scripting. Su simplicidad, sintaxis clara y amplia comunidad lo convierten en una excelente opción para proyectos de investigación y pruebas de concepto. Además, existen muchas bibliotecas relacionadas con criptografía, redes y *blockchain* que facilitan el desarrollo rápido de prototipos funcionales. Empleado como lenguaje principal para la lógica de negocio y los servicios de *backend*.
 - JavaScript: lenguaje de programación utilizado habitualmente para el desarrollo de aplicaciones web. Permite dotar de interactividad y dinamismo a la interfaz de usuario. Empleado para la construcción del cliente web (*frontend*).
- **Tecnologías Web y frameworks**
 - Bootstrap: framework CSS (*Cascading Style Sheets*) que permite desarrollar interfaces web responsivas y adaptativas en poco tiempo, facilitando el diseño de web modernas. Empleado para la construcción del cliente web (*frontend*).

- jQuery: biblioteca JavaScript que simplifica la manipulación del DOM, el manejo de eventos y las solicitudes AJAX (*Asynchronous JavaScript and XML*) en el cliente web. Aunque hay frameworks más modernos, sigue siendo muy útil para tareas ligeras.
- FastAPI: framework asíncrono para la construcción de APIs en Python. Basado en los estándares *OpenAPI* y *JSON Schema*. Está diseñado para ofrecer un alto rendimiento aprovechando la asincronía y el tipado estático. Se ha seleccionada por su capacidad para gestionar múltiples peticiones concurrentes de forma eficiente, integración nativa con *Pydantic* (validación de tipos) y generación automática de documentación interactiva.
- **Criptografía y seguridad**
 - PyNaCl: biblioteca moderna de criptografía en *Python* que ofrece funciones seguras para firmar, cifrar, autenticar y proteger datos sin complicaciones. Es ligera, rápida y fácil de portar a dispositivos IoT.
 - WebCrypto API: estándar que permite realizar operaciones criptográficas de forma segura dentro del navegador: generación de claves, cifrado, descifrado y firma digital. Se utiliza para que el usuario gestione sus claves privadas y cifre sus ficheros localmente, evitando que se expongan datos sensibles al nodo.
 - Pydantic: biblioteca de Python que permite la validación y el modelado de datos mediante anotaciones de tipos. Su principal ventaja radica en que garantiza que los datos cumplan con las estructuras y tipos esperados. Se ha utilizado para definir de forma precisa los modelos de datos que forman parte de la API REST dado que mejora la legibilidad del código, reduce los errores en tiempo de ejecución y facilita la validación automática de entradas / salidas en los *endpoints*.
 - Uvicorn: servidor web ASGI⁴⁴ ligero y de alto rendimiento, diseñado específicamente para ejecutar aplicaciones web asíncronas en Python. Su arquitectura basada en *asyncio* lo hace especialmente adecuado para gestionar

⁴⁴ ASGI o *Asynchronous Server Gateway Interface* es una especificación que define cómo los servidores web deben comunicarse con las aplicaciones web escritas en Python de forma asíncrona. Ideal para aplicaciones en tiempo real y/o de alto rendimiento.

múltiples conexiones simultáneas con baja latencia y consumo mínimo de recursos.

Se usa como servidor de despliegue API REST y para contenido estático.

■ **Red y comunicación distribuida**

- Asyncio: biblioteca estándar de *Python* para programación asíncrona basada en eventos. Esencial para gestionar múltiples conexiones concurrentes entre nodos con un consumo mínimo de recursos.
- Mosquitto: bróker ligero que implementa el protocolo de mensajería MQTT. Diseñado para comunicaciones de tipo publicador/suscriptor en redes de dispositivos de bajo consumo y alta latencia. Especialmente útil en entornos IoT debido a su baja utilización de recursos y su alta eficiencia. Se emplea como mecanismo auxiliar para la distribución de eventos entre nodos.
- Paho-mqtt: biblioteca de *Python* que implementa el protocolo MQTT (cliente). Se usa para permitir a los nodos publicar y suscribirse a eventos a través del bróker MQTT.

■ **Blockchain y consenso**

- Hornet: una de las implementaciones oficiales de nodos de la red IOTA. Escrita en *Go* y diseñada para ofrecer alta eficiencia, bajo consumo de recursos y facilidad de operación. Su arquitectura ligera la hace especialmente adecuada para entornos IoT en proyectos que requieran operar nodos propios de forma autónoma. Se utiliza como nodo IOTA para publicar y recuperar eventos de forma directa. Reduce la latencia y evita la necesidad de depender de nodos públicos.

■ **Infraestructura y virtualización**

- Docker: plataforma de contenedores. Permite empaquetar aplicaciones y sus dependencias, facilitando la portabilidad y despliegue en los nodos distribuidos.
- Armbian: distribución ligera de Linux basada en Debian/Ubuntu. Optimizada para placas de desarrollo (*Raspberry Pi*, *Orange Pi*, etc.).

■ **Almacenamiento**

- SQLite: motor de base de datos relacional embebido, ligero y sin servidor. Ideal para sistemas distribuidos donde se requiere almacenamiento local sencillo.

- Cachetools: biblioteca Python que proporciona estructuras de caché en memoria con distintas políticas de expiración y reemplazo (LRU⁴⁵, LFU⁴⁶ o TTL⁴⁷). Se ha utilizado para mejorar el rendimiento del sistema, evitando accesos repetitivos a recursos de lectura frecuente. Reduce la carga sobre el almacenamiento persistente y mejora la eficiencia general del *backend*.
- **Hardware**
 - Orange Pi One: SBC⁴⁸ (*Single Board Computer*) de bajo consumo y tamaño compacto. Basada en el procesador *Allwinner H3 Quad-Core ARM Cortex-A7*. Dispone de 512MB de RAM DDR3, conectividad *FastEthernet* y puertos de expansión GPIO. Es una alternativa económica y versátil para proyectos IoT y sistemas distribuidos. Compatible con diversas distribuciones Linux, incluyendo *Armbian*. Permite desplegar servicios ligeros en arquitecturas de bajo coste. La **Figura 23** muestra el *cluster* de placas *Orange Pi One* montado para desarrollo y pruebas.

6.1.2. Infraestructura

En este apartado se describen los pasos a seguir para desplegar la infraestructura tecnológica necesaria en el desarrollo, pruebas y despliegue del proyecto. Incluye la instalación de nodos, la configuración de los servicios IOTA / MQTT y los requisitos básicos de los entornos de desarrollo y pruebas.

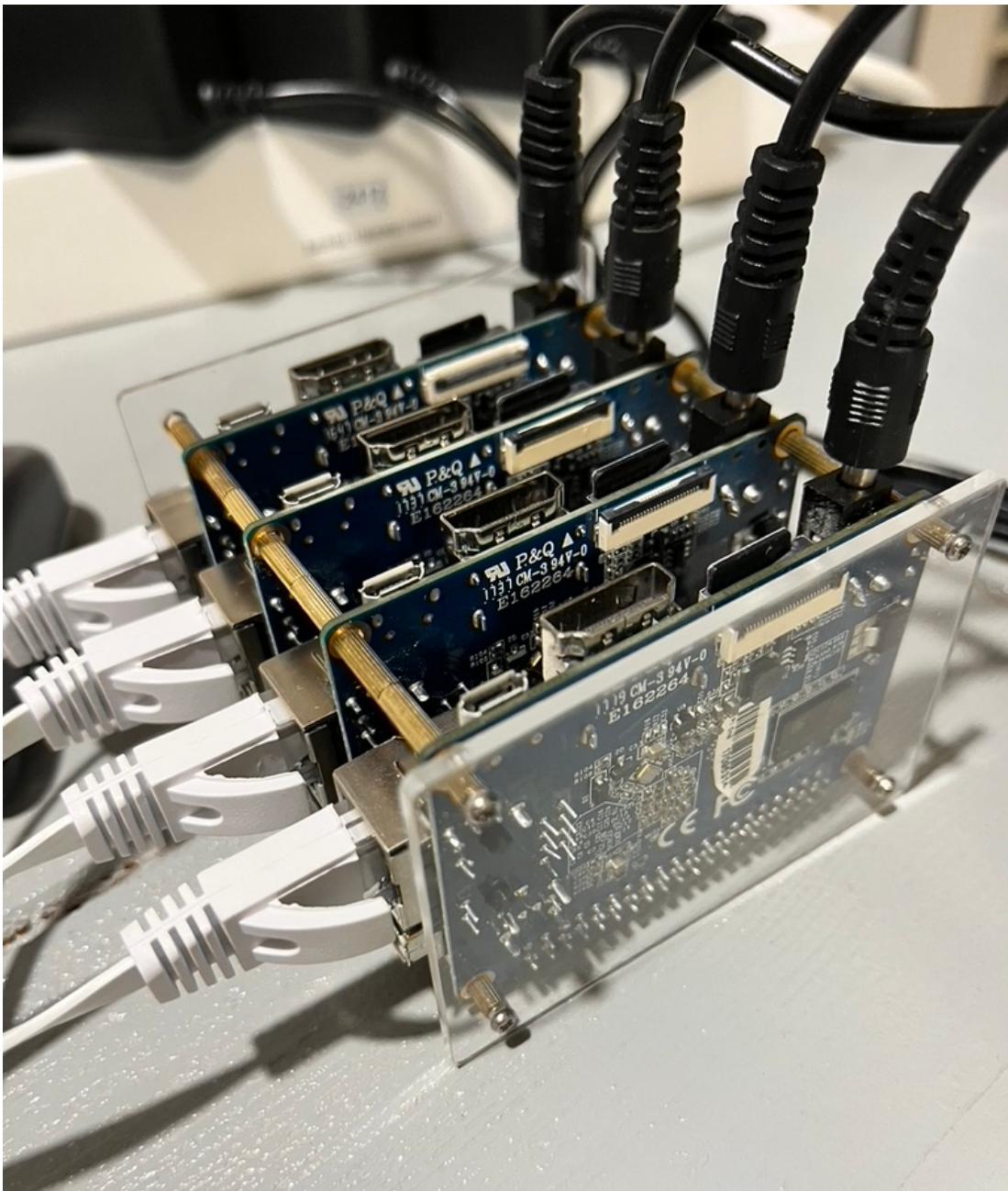
⁴⁵ LRU o *Least Recently Used* es una política de reemplazo de caché que elimina el elemento menos recientemente utilizado cuando la caché alcanza su capacidad máxima bajo el principio de que los datos usados recientemente tienen más probabilidad de ser reutilizados pronto.

⁴⁶ LFU o *Least Frequently Used* es una política de reemplazo de caché que elimina el elemento que ha sido accedido con menor frecuencia bajo el supuesto de que los datos menos utilizados tienen menos probabilidad de ser necesarios en el futuro.

⁴⁷ TTL o *Time To Live* es una política de expiración de caché que asigna un tiempo de vida a cada elemento; una vez transcurrido ese periodo, el elemento se elimina automáticamente, independientemente de su uso.

⁴⁸ SBC o *Single Board Computer* es un ordenador completo integrado en una sola placa, que incluye procesador, memoria, almacenamiento y puertos de entrada/salida. Se utiliza habitualmente en proyectos de electrónica, IoT, automatización y sistemas embebidos por su bajo coste y reducido consumo energético.

Figura 23. Cluster de tarjetas Orange Pi One usado para las pruebas



Fuente: elaboración propia.

6.1.2.1. Entorno virtual de desarrollo y pruebas

Para facilitar el desarrollo y la validación del sistema se ha creado un entorno de trabajo virtualizado. Se describen los pasos seguidos:

1. Montar una máquina virtual sobre *VirtualBox* basada en ARMBIAN para Intel / AMD. Para ello, hay que descargar una imagen de disco desde la página oficial:

https://dl.armbian.com/uefi-x86/Noble_current_minimal

2. La imagen se convierte a formato VDI y se amplía el tamaño de partición a 10 GB:

```
$ VBoxManage convertfromraw Armbian_25.2.3_Uefi-x86_noble_current_6.12.20_minimal.img hdd.vdi --format VDI
$ VBoxManage modifyhd hdd.vdi --resize 10240
```

3. Se crea una nueva máquina virtual y se adjunta la imagen. En este punto, es necesario descargar y montar una ISO auxiliar con la que ampliar el *filesystem* al tamaño máximo de partición. No se puede hacer desde el propio sistema operativo de la máquina virtual, ya que necesita que el *filesystem* esté desmontado:

<https://downloads.sourceforge.net/gparted/gparted-live-1.6.0-3-amd64.iso>

4. Se levanta la máquina virtual y desde la línea de comandos se ejecuta la utilidad de disco gparted:

```
# gparted /dev/sda
> p
> resizepart 3
```

5. Se revisa la integridad del sistema de ficheros y se expande al tamaño de la partición:

```
# e2fsck -f /dev/sda3
# resize2fs /dev/sda3
```

6. Se reinicia la máquina virtual y se desmonta la ISO auxiliar.

```
# reboot
```

7. Se asigna una contraseña inicial de administración y se procede con la creación del usuario de trabajo por defecto (ej. test). La **Figura 24** ilustra el momento en el que el S.O. solicita la creación de una cuenta de usuario sin privilegios (en el ejemplo ‘test’).

Figura 24. Primer acceso a Armbian

```
Creating a new user account. Press <Ctrl-C> to abort
Please provide a username (eg. your first name): test
Create user (test) password: *****
Repeat user (test) password: *****
Please provide your real name: Usuario Test
```

Fuente: elaboración propia.

8. Llegados a este punto, es conveniente habilitar el acceso por SSH a la máquina virtual. En las opciones de *VirtualBox* se configura un puerto de acceso (ej. 2222):

Configuración de red > Configuración NAT > Port forwarding >
127.0.0.1:2222 a 0.0.0.0:22.

9. Se accede por consola mediante SSH y se procede con la actualización de paquetes del sistema operativo y la instalación de librerías / herramientas de desarrollo necesarias:

```
$ sudo -s
# apt-get update
# apt-get upgrade
# apt install -y build-essential libssl-dev pkg-config python3-dev python3-pip
python3-venv python3-requests python3-nacl python3-maturin git libudev-dev
libclang-19-dev mypy python3-dotenv python3-paho-mqtt python3-fastapi python3-
uvicorn python3-cachetools python3-pydantic python3-email-validator python3-httplib
python3-aiofiles ...
```

10. Para descargar la estructura del proyecto en GitHub, se emplea la herramienta git:

```
# cd /opt
# git clone https://github.com/joseigbv/dfs3
```

11. Para subir cambios:

```
# cd dfs3
# git add .
# git commit -m "descripción del commit"
# git push
```

12. Para validar la sintaxis de los nuevos desarrollos se puede usar la herramienta mypy:

```
$ mypy dfs3.py
```

13. Finalmente, se ejecuta el *backend* llamando al script principal dfs3.py:

```
$ python3 dfs3.py
```



dfs3 0.1 - Distributed File Storage System for IoT with Blockchain
Author: José Ignacio Bravo Vicente <nacho.bravo@gmail.com>

[LOG] Starting dfs3 system...

```
[LOG] Database 'data/dfs3.db' already exists
[LOG] Loading node config...
Enter passphrase to decrypt private key: [secret]
[LOG] Starting MQTT listener...
[LOG] Starting API REST listener...
[LOG] Node ID: 52a5e6bfe59f0e4404ddc3216700b2b3a362aceeaaf8ec30534ba863db84f1819
loaded and ready
...
```

6.1.2.2. Instalación y configuración de Hornet

Se detalla el proceso de instalación y configuración del nodo *Hornet*, el software oficial ligero para operar como nodo en la red IOTA. Para proceder con la instalación y configuración del *docker* se han seguido los siguientes pasos:

1. Descarga e instalación del contenedor oficial desde la página de IOTA:

```
# mkdir node-docker-setup && cd node-docker-setup && curl -L https://node-docker-
setup.iota.org/iota-testnet | tar -zx
```

2. Es necesario modificar su configuración para que realice PoW por los nodos:

```
# cp env_template .env
# vi .env
...
HTTP_PORT=8080
...

# vi config.json
...
  "restAPI": {
    "pow": {
      "enabled": true
    }
  }
...
```

3. Ejecución del *docker*:

```
# docker-compose run hornet tools pwd-hash
...
Enter a password:
Re-enter your password:

Success!
...

# ./prepare_docker.sh
# docker compose up -d
```

4. Verificación de configuración y funcionamiento:

```
# curl -v http://localhost/api/core/v2/info
{
  "name": "HORNET",
  "version": "2.0.2",
  "status": {
    "isHealthy": true,
    "latestMilestone": {
      "index": 6356151,
      "timestamp": 1745517068,
      ...
    }
  },
  ...
  "features": [
    "pow"
  ]
}
...
```

5. Para depuración, monitorización y parada del servicio:

```
# docker compose logs -f hornet
# docker compose down
```

6.1.2.3. Servidor MQTT

Los pasos seguidos para desplegar el bróker MQTT *Mosquitto* han sido los siguientes:

1. Instalación y configuración de *Mosquitto* como bróker MQTT con persistencia:

```
# apt-get install mosquitto
# vi /etc/mosquitto/mosquitto.conf
...
listener 1883
allow_anonymous true
persistence true
persistence_location /mosquitto/data/
...
```

2. Arranque del servicio:

```
# service mosquitto start
```

3. Para probar que todo funciona correctamente, abrir dos terminales, suscribir en la primera y ejecutar una prueba de publicación la segunda:

```
$ mosquitto_pub -h mqtt.dfs3.net -p 1883 -t dfs3/events -m '{"block_id": "0x8e113d406b0a907384fd5f0eff2356f6420375a626b71b6e398fedc5133e689a", "event_type": "node_registered", "timestamp": "2025-05-01T16:33:56.447421+00:00", "node_id": "a80da468fb905f80915056a735a55720b64537bfd328f66ac34b6e2af6056c60"}'
$ mosquitto_sub -h mqtt.dfs3.net -p 1883 -t dfs3/events
```

```

...
{"block_id": "0x8e113d406b0a907384fd5f0eff2356f6420375a626b71b6e398fedc5133e689a",
 "event_type": "node_registered", "timestamp": "2025-05-01T16:33:56.447421+00:00",
 "node_id": "a80da468fb905f80915056a735a55720b64537bfd328f66ac34b6e2af6056c60"}
...

```

4. Para configurar el soporte TLS, editar el fichero de configuración y reiniciar el servicio:

```

# vi /etc/mosquitto/mosquitto.conf
...
# Listener para conexiones seguras
listener 8883
cafile /etc/mosquitto/certs/fullchain.pem
certfile /etc/mosquitto/certs/cert.pem
keyfile /etc/mosquitto/certs/privkey.pem
...
# service mosquitto restart

```

6.1.2.4. Creación y renovación de certificados

Para garantizar la seguridad de las comunicaciones entre los distintos componentes, se ha optado por el uso de los certificados TLS emitidos por *Let's Encrypt*. Se describe el proceso de solicitud, renovación e instalación de estos certificados usando la herramienta `certbot`.

Para facilitar el desarrollo y pruebas del proyecto, se ha registrado un nuevo nombre de dominio (`dfs3.net`). La **Tabla 14** muestra la lista de certificados necesarios para el desarrollo y testeo del prototipo:

Tabla 14. Listado de certificados TLS necesarios

Componente	Hostname	Uso	Certificado
Interfaz web	node.dfs3.net	Acceso seguro al <i>frontend</i> de cada nodo.	Certificado TLS (Let's Encrypt)
API REST	node<n>.dfs3.net	Comunicación segura entre cliente web y <i>backend</i> REST y <i>frontend</i> local.	Certificado TLS (Let's Encrypt por nodo)
Nodo IOTA	iota.dfs3.net	Acceso a nodo IOTA para publicación de mensajes.	Certificado TLS (Let's Encrypt)
Bróker MQTT	mqtt.dfs3.net	Cifrado de mensajes MQTT entre nodos.	Certificado TLS (Let's Encrypt)

Fuente: elaboración propia.

El siguiente ejemplo muestra una solicitud de certificado para node0.dfs3.net⁴⁹:

```
# certbot certonly --standalone -d node0.dfs3.net
Saving debug log to /var/log/letsencrypt/letsencrypt.log
Requesting a certificate for node0.dfs3.net

Successfully received certificate.
Certificate is saved at: /etc/letsencrypt/live/node0.dfs3.net/fullchain.pem
Key is saved at:          /etc/letsencrypt/live/node0.dfs3.net/privkey.pem
This certificate expires on 2025-09-16.
These files will be updated when the certificate renews.

...
```

Para la renovación del certificado, ejecutar:

```
# certbot renew --cert-name node0.dfs3.net
```

Para configurar el *backend*:

```
# ln -s /etc/letsencrypt/live/node0.dfs3.net/fullchain.pem /opt/dfs3/data/
# ln -s /etc/letsencrypt/live/node0.dfs3.net/privkey.pem /opt/dfs3/data/
```

6.1.2.5. Instalación y configuración de nodos

El proceso del software base es similar al del nodo virtual de desarrollo.

1. Descargar una imagen del S.O. para los nodos IoT desde la URL oficial de *Armbian*, tal y como se indica en la **Figura 25**: <https://www.armbian.com/orange-pi-one/>

Figura 25. Descarga de S.O. de los nodos IoT

Distro	Variant	Extensions	Torrent	Integrity	Size
 Debian 12 (Bookworm)	Minimal / IOT			SHA ASC	236.6 MB
 Ubuntu 24.04 (Noble)	Minimal / IOT			SHA ASC	227.3 MB
 Debian Testing (Trixie)	Minimal / IOT			SHA ASC	257.1 MB

* Minimal images have very small footprint. They come only with essential packages and build-in systemd-networkd. They are optimised for automation and production deployments.

Fuente: elaboración propia.

⁴⁹ Para automatizar la generación de certificados, el puerto TCP/80 debe ser accesible desde Internet.

2. Descomprimir y grabar la imagen a una tarjeta MicroSD:

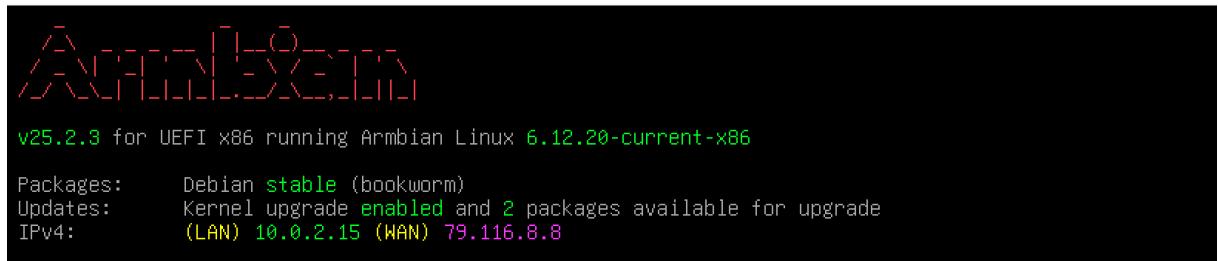
```
# dd if=Armbian_25.5.1_Orangepine_noble_current_6.12.23_minimal.img  
of=/dev/rdisk2 bs=4m
```

3. Conectar y configurar el puerto serie (RX -> TX, TX -> RX, GND y 5V). Acceder por terminal.

La primera vez, nos pedirá crear la password de *root* y un usuario sin privilegios (ej. *dfs3*).

```
$ screen /dev/tty.SLAB_USBtoUART 115200  
...
```

Figura 26. Pantalla principal Armbian



Fuente: elaboración propia.

4. Configurar la red, actualizar el sistema operativo, cambiar el *hostname* y reiniciar:

```
# armbian-config  
# armbian-upgrade  
# hostnamectl hostname node1  
# timedatectl set-timezone Europe/Madrid  
# reboot
```

5. Acceder por SSH e instalar las herramientas y dependencias necesarias:

```
# apt install screen git python3 python3-requests python3-nacl python3-maturin  
python3-dotenv python3-paho-mqtt python3-fastapi python3-unicorn python3-  
cachetools python3-pydantic python3-email-validator python3-httplib python3-aiofiles  
python3-multipart
```

6. Generar un certificado con certbot. Dado que tenemos acceso a la zona, en esta ocasión usaremos el desafío por DNS:

```
$ certbot --cert-name node1.dfs3.net --manual --preferred-challenges dns certonly  
-d node1.dfs3.net -d node.dfs3.net
```

7. Finalmente, instalar el software, configurar el certificado y ejecutar el *backend*:

```
# cd /opt
```

```
# git clone https://github.com/joseigbv/dfs3
# cd dfs3
# ln -s /etc/letsencrypt/live/node1.dfs3.net/privkey.pem data/privkey.pem
# ln -s /etc/letsencrypt/live/node1.dfs3.net/fullchain.pem data/fullchain.pem
$ ./dfs3.py
```

6.2. Desarrollo e implementación del servicio

Se describe en detalle el proceso de implementación del servicio, incluyendo su estructura interna, componentes y lógica de funcionamiento.

6.2.1. Estructura del proyecto

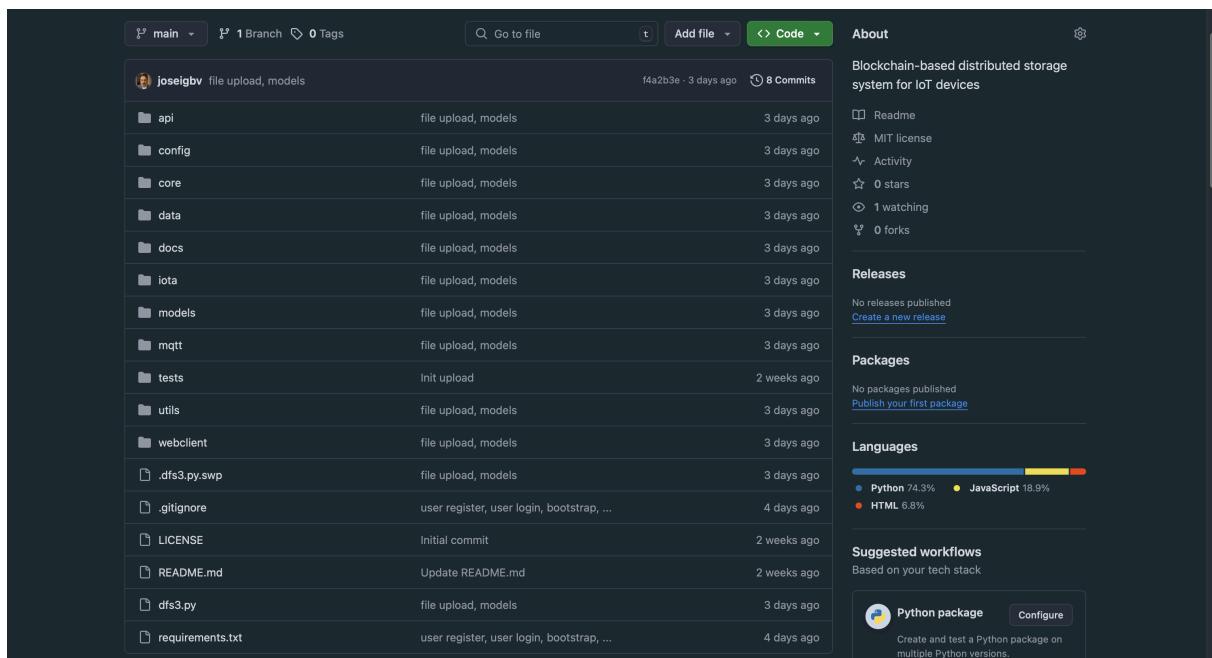
Para facilitar su mantenimiento y escalabilidad, se ha seguido una arquitectura modular. La raíz del proyecto contiene el ejecutable principal (`dfs3.py`), encargado de inicializar el nodo y lanzar los servicios de *backend*: *listener API REST* y *manejador de eventos MQTT*. Se han definido varias carpetas para agrupar los módulos según su funcionalidad:

- `core/`: contiene la lógica interna del nodo, incluyendo la generación y gestión de claves, el procesamiento de eventos, el control de estado y la interacción con el almacenamiento local.
- `api/`: implementa la API REST utilizando *FastAPI*. Se divide en módulos como `server.py`, que define y lanza el servidor web o `routes.py`, donde se agrupan los diferentes *endpoints*.
- `webclient/`: almacena los archivos estáticos que constituyen el cliente web (HTML, CSS, JavaScript, imágenes).
- `config/`: parametrización del sistema con ficheros auxiliares como `node.json`, que guarda la configuración del nodo, incluyendo su clave pública, alias y demás metadatos.
- `data/`: contiene los archivos de datos persistentes: base de datos SQLite, registros de eventos, carpetas y ficheros de usuario, ficheros de metadatos, ficheros de datos, etc. Sirve como punto central para el almacenamiento local en el *backend* del nodo.
- `models/`: agrupa los modelos y tipos de datos definidos mediante *Pydantic*. Incluye las estructuras que representan usuarios, nodos, ficheros, eventos y otros elementos

clave utilizados en la API y lógica interna. Aseguran la validación, el tipado y la coherencia de los datos intercambiados.

- `utils/`: contiene funciones auxiliares y utilidades comunes reutilizadas en los distintos módulos del proyecto: funciones de cifrado, generación de identificadores, manejo de fechas, etc. Su propósito es mantener el código modular y evitar duplicidades.
- `iota/`: agrupa los módulos relacionados con la interacción con la red IOTA, incluyendo la publicación y recuperación de eventos desde la *Tangle*.
- `mqtt/`: contiene los módulos encargados de la comunicación a través del protocolo MQTT. Incluye la lógica para publicar y suscribirse a mensajes, gestionar la conexión con el bróker y procesar los eventos entrantes.

Figura 27. Estructura de carpetas `dfs3` en GitHub



Fuente: elaboración propia.

La **Figura 27** muestra la estructura del repositorio principal y como se organiza el código fuente en módulos funcionales. El uso de *GitHub* facilita el mantenimiento del código y su despliegue en dispositivos IoT.

6.2.2. Implementación de API REST

El sistema expone una interfaz *RESTful* que permite a los usuarios autenticarse, gestionar sus archivos o acceder a los nodos distribuidos. La API se ha desarrollado en *Python* utilizando *FastAPI*.

Figura 28. Interfaz OpenAPI autogenerada

The screenshot shows the FastAPI documentation interface for an OpenAPI specification. At the top, it says "FastAPI 0.1.0 OAS 3.1" and has a link to "/openapi.json". Below this, there's a section titled "default" which lists several API endpoints:

- POST /api/v1/auth/challenge** Request Challenge
- POST /api/v1/auth/register** Register
- POST /api/v1/auth/verify** Verify
- GET /api/v1/files** Get Files
- POST /api/v1/files** Upload File
- GET /api/v1/files/{file_id}** Download File

Fuente: elaboración propia.

La **Figura 28** muestra parte de la especificación de la API REST generada automáticamente con *FastAPI*. Para acceder a la información completa, se puede consultar el anexo “[API REST](#)” con una descripción detallada de cada *endpoint*.

6.2.2.1. Listado de endpoints

La **Tabla 15** contiene el listado de *endpoints* implementados. La lógica de enrutamiento se ha estructurado usando el componente *API Router*. Se separan los *endpoints* en conjuntos funcionales:

- **Autenticación y registro:** basada en criptografía asimétrica, sin almacenamiento de contraseñas. Los usuarios se autentican firmando un reto aleatorio (*challenge*) que demuestra la posesión de la clave privada asociada al identificador de usuario.
- **Gestión de usuarios:** permite listar y consultar información básica sobre usuarios registrados, incluyendo su identificador único, alias y clave pública (necesarios para la compartición cifrada de archivos).

- **Gestión de nodos:** expone información básica (identificador y alias) de cada nodo. La sincronización y el estado se gestionan fuera de la API mediante eventos MQTT.
- **Gestión de ficheros:** para subir ficheros, gestionar su representación en el espacio del usuario (renombrado, copiado y borrado), compartir, acceder a su contenido y metadatos, etc.

Tabla 15. Listado de los endpoints en la API REST

Categoría	Método	Endpoint	Descripción
Autenticación	POST	/auth/register	Registro de nuevo usuario.
Autenticación	POST	/auth/challenge	Solicitud de reto para <i>login</i> .
Autenticación	POST	/auth/verify	Verificación de reto.
Usuarios	GET	/users	Listar usuarios registrados.
Usuarios	GET	/users/{user_id}	Obtener información de un usuario.
Nodos	GET	/nodes	Listar nodos conocidos.
Nodos	GET	/nodes/{node_id}	Información detallada de un nodo.
Ficheros	POST	/files	Subida de fichero cifrado.
Ficheros	GET	/files	Listar las entradas virtuales del usuario autenticado.
Ficheros	GET	/files/{filename}	Descargar el fichero cifrado (incluye información criptográfica en cabeceras). Si el nodo no almacena el fichero de datos, consulta a otros nodos y actúa como proxy.
Ficheros	GET	/files/{file_id}/data	Descargar el contenido cifrado. Usado para replicación.
Ficheros	GET	/files/{file_id}/meta	Obtener metadatos.
Ficheros	PATCH	/files/{filename}	Renombrar entrada virtual.
Ficheros	DELETE	/files/{filename}	Eliminar entrada virtual y si aplica, fichero de datos.
Ficheros	POST	/files/share	Compartir fichero con otro usuario.

Fuente: elaboración propia.

6.2.2.2. Organización modular

La estructura modular del sistema se ha diseñado para favorecer la escalabilidad, el mantenimiento y la reutilización del código. Cada módulo encapsula una funcionalidad específica. La **Tabla 16** contiene un listado con los principales módulos implementados.

Tabla 16. Estructura de los principales módulos del backend

Módulo	Propósito
<code>api/routes/</code>	Define los <i>endpoints</i> (<code>auth</code> , <code>files</code> , <code>users</code> , <code>nodes</code> , <code>status</code> , ...).
<code>api/models/</code>	Contiene los modelos <i>Pydantic</i> para validar la entrada y salida de datos.
<code>config/settings.py</code>	Configuración global del sistema, incluyendo puertos, rutas y opciones de debug.
<code>core/auth.py</code>	Lógica de autenticación basada en firma de retos y tokens.
<code>core/context.py</code>	Variables globales como la configuración del nodo o su clave privada desbloqueada.
<code>core/entries.py</code>	Gestión de las entradas virtuales (visibilidad de archivos).
<code>core/events.py</code>	Publicación de eventos firmados a IOTA y notificación vía MQTT.
<code>core/event_handler.py</code>	Procesamiento de eventos recibidos: actualización de estado, replicación, etc.
<code>core/files.py</code>	Lógica de acceso al sistema de ficheros local (<code>.storage/</code> , <code>.meta/</code>).
<code>core/nodes.py</code>	Gestión del estado de los nodos.
<code>core/users.py</code>	Gestión de usuarios: alta, búsqueda, claves públicas, sincronización.
<code>iota/client.py</code>	Cliente de conexión con nodo IOTA para publicación y recuperación de eventos.
<code>models/</code>	Estructuras de datos compartidas entre módulos internos.
<code>mqtt/client.py</code>	Cliente MQTT para publicar mensajes al bróker.
<code>mqtt/listener.py</code>	Suscriptor MQTT que reacciona ante eventos publicados por otros nodos.
<code>utils/crypto.py</code>	Funciones criptográficas auxiliares: firma, cifrado simétrico/asimétrico.
<code>utils/db.py</code>	Funciones auxiliares para el acceso a la base de datos SQLite.
<code>utils/system.py</code>	Utilidades generales del sistema: lectura de ficheros, logs, comprobaciones, etc.

Fuente: elaboración propia.

6.2.2.3. Ejemplo de definición de ruta

Se adjunta un ejemplo simplificado del *endpoint* GET `/files/{file_id}` que permite a un usuario autenticado con permisos de acceso, descargar un fichero cifrado:

```
@router.get("/files/{file_id}")
async def download_file(
    user_id: str = Depends(require_auth),
    file_id: constr(regex=r"^[a-f0-9]{64}$") = Path(...):
):
```

```
if not user_has_access(user_id, file_id):
    raise HTTPException(status_code=403, detail="Access denied")

file_path = os.path.join(STORAGE_DIR, file_id)
if not os.path.isfile(file_path):
    raise HTTPException(status_code=404, detail="File not found")

return FileResponse(path=file_path, media_type="application/octet-stream")
```

Este ejemplo demuestra varios principios aplicados:

- Autenticación con `Depends(require_auth)`, usando un token temporal generado aleatoriamente y validado mediante firma asimétrica.
- Validación de entrada con `constr(regex=...)` para asegurar que `file_id` es un hash SHA-256 válido.
- Respuestas seguras con `FileResponse`, sin cargar el archivo entero en memoria (*streaming*).

6.2.2.4. Validación de entrada/salida

Se han definido tipos y usado modelos *Pydantic* para validar las entradas al *backend*, como el que se muestra en el siguiente ejemplo:

```
class UploadFileMetadata(StrictBaseModel):
    """
    Represents the metadata required when uploading an encrypted file.
    This model is used to validate and structure the JSON metadata provided
    in the multipart/form-data request.
    """

    filename: constr(regex=RE_FILENAME)
    file_id: constr(regex=RE_FILE_ID)
    size: conint(ge=0, le=MAX_FILE_SIZE)
    mimetype: constr(regex=RE_MIMETYPE)
    sha256: constr(regex=RE_FILE_ID)
    iv: constr(regex=RE_BASE64)
    tags: List[str] = []
    authorized_users: List[AuthorizedUserEntry]
    ...
```

6.2.2.5. Flujo simplificado para el endpoint POST /files

A modo de ejemplo, se incluye el flujo de subida de un fichero:

1. El cliente sube el archivo cifrado y un JSON con sus metadatos.

2. El nodo valida que el usuario está autenticado, el hash coincide con el contenido (`file_id`) y la estructura de metadatos es correcta.
3. Guarda el fichero en `.storage/<file_id>.dat`.
4. Publica un evento `file_created` firmado digitalmente en IOTA que contiene los metadatos y avisa por mensaje MQTT.
5. Los nodos reciben el aviso y descargan el evento de la red IOTA con los metadatos.
6. Cada nodo se sincroniza generando un fichero `.meta/<file_id>.json` y creando una entrada virtual en el espacio del usuario.

6.2.2.6. Servidor de contenido estático

Adicionalmente, la aplicación sirve contenido estático (cliente web) desde un directorio dedicado (`/webclient`). Esto permite integrar el acceso a la interfaz de usuario (HTML + JS) y a la API en un único proceso.

6.2.3. Gestión de almacenamiento

La gestión de almacenamiento se basa en el uso del *filesystem* de cada nodo para guardar tanto los ficheros cifrados como sus metadatos. El almacenamiento se organiza en tres carpetas principales dentro de `data`:

- `.storage/`: almacena el contenido cifrado de cada fichero, identificado por su `file_id`.
- `.meta/`: contiene un fichero JSON por cada `file_id`. Incluye los metadatos asociados al archivo de datos: tamaño, propietario, usuarios autorizados, claves cifradas, tags, ...
- `.users/`: el listado y la visualización de ficheros a los que cada usuario tiene acceso se gestiona creando entradas virtuales que apuntan al fichero de metadatos.

La **Figura 29** muestra un caso concreto: debajo del directorio `.users/` existe una carpeta para cada usuario, identificada por su `user_id`, que contiene la lista de entradas virtuales a las que tiene acceso (ej. `test.pdf`). En `.storage/` se almacena el contenido real de cada fichero identificado por `<file_id>.dat`. Finalmente, `.meta/` contiene la lista de ficheros `<file_id>.json` con sus metadatos asociados.

Figura 29. Ejemplo de estructura de ficheros dfs3

```
.users/fd8534506ad3b8a7262a3f82d28f28c70f72d1b4d74e401b28342d73b70a870d  
.users/fd8534506ad3b8a7262a3f82d28f28c70f72d1b4d74e401b28342d73b70a870d/test.txt  
.users/fd8534506ad3b8a7262a3f82d28f28c70f72d1b4d74e401b28342d73b70a870d/Logo_UNIR.png  
.users/fd8534506ad3b8a7262a3f82d28f28c70f72d1b4d74e401b28342d73b70a870d/dfs3.png  
...  
.storage/25ab5f4ee0e9dcf83561a4e92051bfcc8343678ab2ac0fe0513933821f9b0486.dat  
.storage/54347bec5b5c12a856716d6ab846f56a26c30a791c408dea4b4d55e1ee3e6022.dat  
.storage/d41204ba37ffc917d4c6efb4f05d2e5b36be8fb3a2b04604f81261099798c262.dat  
.storage/278caf41604ee0c494624a0fe58b5411477463cf9d3d84ba572a34219f23605.dat  
.storage/61aee793c4001ee4372a914cffa7288c759cc0a2f9990fa4275566bf8198dea8.dat  
...  
.meta/25ab5f4ee0e9dcf83561a4e92051bfcc8343678ab2ac0fe0513933821f9b0486.json  
.meta/54347bec5b5c12a856716d6ab846f56a26c30a791c408dea4b4d55e1ee3e6022.json  
.meta/d41204ba37ffc917d4c6efb4f05d2e5b36be8fb3a2b04604f81261099798c262.json
```

Fuente: elaboración propia.

En la **Figura 30** podemos observar como el fichero test.pdf y d41204ba37ffc917d4c6ef... comparten el mismo inodo (1989716). Ambas entradas apuntan a un mismo contenido.

Figura 30. Ejemplo de entrada virtual en dfs3

```
$ ls -li .users/9b18430e56deaedfbf3c5a0d9d8493c2f499ef841842cb820a27fd44bd993cbb/test.pdf  
1989716 -rw-r--r-- 2 root root 802 May 12 11:21 .users/9b18430e56deaedfbf3c5a0d9d8493c2f49  
$ ls -li .meta/d41204ba37ffc917d4c6efb4f05d2e5b36be8fb3a2b04604f81261099798c262.json  
1989716 -rw-r--r-- 2 root root 802 May 12 11:21 .meta/d41204ba37ffc917d4c6efb4f05d2e5b36be
```

Fuente: elaboración propia.

La **Figura 31** muestra el contenido de una entrada virtual con los metadatos del fichero.

Figura 31. Ejemplo de entrada virtual con metadatos

```
$ more .users/9b18430e56deaedfbf3c5a0d9d8493c2f499ef841842cb820a27fd44bd993cbb/test.pdf  
{  
    "file_id": "d41204ba37ffc917d4c6efb4f05d2e5b36be8fb3a2b04604f81261099798c262",  
    "filename": "test.pdf",  
    "owner": "9b18430e56deaedfbf3c5a0d9d8493c2f499ef841842cb820a27fd44bd993cbb",  
    "size": 12093,  
    "iv": "voZmdTyKcsuaTwp9",  
    "sha256": "9c23487475fdb538d72bbd78bdd57206b9379ead2e6d9be90c20a0f7d5d51aed",  
    "mimetype": "application/pdf",  
    "tags": [  
        "test",  
        "dfs3"  
    ],  
    "authorized_users": [  
        {  
            "user_id": "9b18430e56deaedfbf3c5a0d9d8493c2f499ef841842cb820a27fd44bd993cbb",  
            "encrypted_key": "ZB2xXoQebjpry1uLloL50U7cX9hLEqKQPU4kDtUrc08F8qSorkErX1EilwYGTGfC",  
            "iv": "gAFhD2zH4mm64Mf"  
        }  
    ]  
}
```

Fuente: elaboración propia.

6.2.4. Gestión de eventos distribuidos

El sistema utiliza un modelo *event-driven* distribuido para coordinar la sincronización entre nodos, tal y como se describe a continuación.

6.2.4.1. Estructura general

El código para la gestión de eventos se ha estructurado en módulos, tal y como muestra la **Tabla 17**.

Tabla 17. Estructura modular del sistema de gestión de eventos

Módulo	Responsabilidad
<code>core/events.py</code>	Construcción, firma y publicación de eventos en IOTA y MQTT
<code>core/iota.py</code>	Envío de eventos a la Tangle como bloques JSON
<code>core/mqtt.py</code>	Publicación del <code>block_id</code> vía bróker <i>Mosquitto</i>
<code>core/context.py</code>	Acceso seguro a la clave privada del nodo para firmar eventos
<code>core/event_handler.py</code>	Subsistema de escucha y respuesta a eventos (crear entrada, etc.)

Fuente: elaboración propia.

6.2.4.2. Formato del evento

Todos los eventos se definen como documentos JSON estructurados y firmados, siguiendo el esquema descrito en la **Tabla 18**. A continuación, un extracto de un evento tipo `file_created`:

```
{  
    "type": "file_created",  
    "timestamp": "2025-05-11T13:45:00Z",  
    "node_id": "abc123...",  
    "protocol": "dfs3/1.0",  
    "payload": {  
        "file_id": "a1b2c3...",  
        "owner": "e638b7...",  
        "size": 123456,  
        "mimetype": "application/pdf",  
        ...  
    },  
    "signature": "MEUCIQD+ePY..."  
}
```

Tabla 18. Estructura general de un evento

Campo	Tipo	Descripción
event_type	string	Tipo de evento: node_registered, file_created, etc.
timestamp	string (ISO 8601)	Fecha y hora UTC de creación del evento en formato ISO 8601.
node_id	string (SHA-256)	Identificador del nodo que ha emitido el evento (hash SHA256 de su clave pública).
protocol	string	Versión del protocolo (ej. dfs3/1.0).
payload	object	Contenido específico del evento. Varía según event_type.
signature	string (base64)	Firma digital (nodo) del campo payload en formato ED25519.

Fuente: elaboración propia.

6.2.4.3. Publicación IOTA / MQTT

El flujo para la publicación de un evento es el siguiente:

1. El evento se serializa a formato JSON y se firma digitalmente.
2. Se publica como un bloque de datos en IOTA con persistencia descentralizada.
3. El block_id resultante se publica a un canal común en MQTT.
4. Los demás nodos, al recibir el block_id, consultan IOTA, recuperan el evento completo y lo procesan localmente.

Figura 32. Evento publicado en la red IOTA

The screenshot shows the IOTA Explorer interface with the following details:

- Header:** EXPLORER, Explorer, Visualizer, Statistics, EVM, Search bar (Search the tangle...), USD, NETWORK IOTA Testnet.
- Block Details:**
 - Data Block** (with a blue info icon)
 - Status: Confirmed
 - Referenced by Milestone 6391210 - Saturday, April 26, 2025 9:07 PM - a few seconds ago
 - General tab selected
 - Block ID: 0xffffe72e609c179528408c657ebedcf4043de179b767c3e760e2bf2bcd1105267

Fuente: elaboración propia.

La **Figura 32** muestra un bloque de datos confirmado en la red *Testnet* de IOTA. Este tipo de registros son sin cargo y permiten almacenar eventos de forma inmutable.

6.2.4.4. Sincronización inicial

Se ha implementado un mecanismo de sincronización para cuando se incorpora un nuevo nodo a la red. Durante el proceso de registro, se ejecuta una llamada al *endpoint GET /api/v1/events* del nodo 0, que proporciona un histórico de los eventos publicados hasta la fecha. Con esta información, el nuevo nodo puede:

1. Descargar el contenido completo de cada evento desde IOTA (usando su `block_id`).
2. Procesar localmente los eventos relevantes para sincronizar su estado inicial: ficheros creados o compartidos (`file_created`, `file_shared`), cambios estructurales en la red (`node_registered`, `user_created`), etc.
3. Una vez completada la sincronización, el nodo se suscribe al canal MQTT (`dfs3/events`) para comenzar a recibir eventos en tiempo real al igual que el resto de nodos.

6.2.5. Uso de cachés y optimizaciones en backend

Con el objetivo de mejorar el rendimiento y reducir la carga en operaciones repetitivas, se ha incorporado una caché que permite almacenar temporalmente las respuestas a consultas frecuentes y/o costosas. Gracias a esta capa de optimización, el *backend* puede mantener un rendimiento aceptable, incluso ante escenarios de concurrencia moderada.

Para su implementación se ha optado por una estrategia basada en `cachetools.TTLCache`, combinada con almacenamiento persistente en SQLite. Se ha utilizado en operaciones como la recuperación de la clave pública de usuarios y nodos o la validación de tokens de sesión activos. Se adjunta un extracto de código que ilustra el uso de esta caché:

```
# cache de 100 elementos, se elimina el más antiguo
_user_cache: LRUCache[str, UserEntry] = LRUCache(maxsize=100)
...
@cached(_user_cache, key=lambda user_id: user_id)
def get(user_id: str) -> UserEntry | None:
    """
    Retrieves a user by user_id from cache or database.
    """
    with sqlite3.connect(DB_FILE) as conn:
        conn.row_factory = sqlite3.Row
        with closing(conn.cursor()) as cursor:
            cursor.execute("""
                SELECT user_id, alias, public_key
            ...

```

6.3. Implementación del cliente web

La aplicación web permite al usuarios interactuar con el sistema de almacenamiento distribuido desde un navegador. Se ha desarrollado con tecnologías estándar: HTML5, CSS, JavaScript, Bootstrap y jQuery. Toda la lógica criptográfica se ha implementado en el lado cliente usando las librerías *@noble/ed25519* y *libsodium-wrappers*.

Figura 33. Cliente web: listado de ficheros

The screenshot shows the DFS3 web application interface. At the top, there is a header with the logo 'dfs3' and the text 'Mis archivos'. On the right side of the header, it says 'Hola, nacho' and has a 'Cerrar sesión' button. Below the header is a blue button labeled 'Subir archivo'. The main area is a table listing five files:

Nombre	Tamaño	Fecha de Creación	Acciones	Tipo
test.txt	19 B	12 may 2025, 20:41:55	[Compartir] [Renombrar] [Copiar] [Borrar]	
test (1).pdf	11.8 KB	12 may 2025, 20:42:23	[Compartir] [Renombrar] [Copiar] [Borrar]	
test.pdf	11.8 KB	12 may 2025, 20:39:28	[Compartir] [Renombrar] [Copiar] [Borrar]	
Logo_UNIR.png	16.8 KB	12 may 2025, 20:40:31	[Compartir] [Renombrar] [Copiar] [Borrar]	
1234.txt	5 B	12 may 2025, 20:41:42	[Compartir] [Renombrar] [Copiar] [Borrar]	

At the bottom of the page, there is a footer with the text 'TFG: Sistema Blockchain de Almacenamiento Distribuido para Dispositivos IoT' and 'Universidad Internacional de La Rioja (UNIR) · José Ignacio Bravo <nacho.bravo@gmail.com> · 2025'.

Fuente: elaboración propia.

La **Figura 33** muestra la pantalla principal. Esta vista facilita la gestión segura de los contenidos almacenados en la red descentralizada: subir y descargar archivos, compartirlos, renombrarlos, etc.

6.3.1. Estructura de archivos

La **Tabla 19** muestra la estructura de ficheros que componen la aplicación cliente:

Tabla 19. Contenido de la carpeta 'webclient'

Archivo	Descripción
<code>login.html</code>	Página de <i>login</i> , permite seleccionar las claves de usuario y autenticarse.
<code>login.js</code>	Lógica de <i>login</i> mediante desafío/respuesta con firma digital.
<code>register.html</code>	Formulario de alta de usuario: alias, nombre, email y contraseña.
<code>register.js</code>	Lógica de registro de usuario: generación de claves, cifrado y envío a la API.

index.html	Listado de archivos disponibles, con opción de descarga y manipulación.
index.js	Recuperación de lista de ficheros, generación de botones de descarga, ...
upload.html	Vista para seleccionar y subir ficheros.
upload.js	Cifrado del fichero, generación de metadatos y envío a la API.
common.js	Módulo con funciones comunes: utilidades, acceso API.
crypto.js	Módulo con funciones específicas de cifrado y descifrado.

Fuente: elaboración propia.

6.3.2. Generación y gestión de claves en el navegador

El navegador genera, cifra y almacena las claves del usuario. El identificador único de usuario (`user_id`) se deriva como un hash SHA-256 de su clave pública. El siguiente fragmento de código muestra como generar el par de claves y derivar su `user_id`:

```

...
const keyPair = await ed.generateKeyPair();
const user_id = await sha256ToHex(keyPair.publicKey);
...

```

6.3.3. Registro de usuarios

El alta de usuario se realiza a través de la URL `/register.html`. Para registrar un usuario, el navegador ejecuta los siguientes pasos:

1. Sigue la URL `/register.html`.
2. Sigue la URL `/register.html`.
3. Sigue la URL `/register.html`.
4. Sigue la URL `/register.html`.
5. Sigue la URL `/register.html`.
6. Sigue la URL `/register.html`.

A continuación, se muestra un extracto del código de derivación y cifrado de clave:

```

const passwordKey = await crypto.subtle.importKey("raw", encodedPassword,
    "PBKDF2", false, ["deriveKey"]);
const aesKey = await crypto.subtle.deriveKey({ name: "PBKDF2", ... }, passwordKey, {
    name: "AES-GCM", length: 256 }, false, ["encrypt"]);

```

```
const encryptedPrivateKey = await crypto.subtle.encrypt({ name: "AES-GCM", iv },  
aesKey, privateKey);  
...
```

Por seguridad, no se envían la clave privada ni sus derivados al servidor. Se realiza un POST /users con los siguientes datos:

```
{  
  "user_id": "...",  
  "alias": "...",  
  "name": "...",  
  "email": "...",  
  "public_key": "..."  
}
```

Figura 34. Cliente web: registro de usuarios

The screenshot shows a web-based user registration form titled 'Registro de usuario'. The form includes fields for 'Alias*' (filled with 'nacho'), 'Nombre completo*' (filled with 'José Ignacio Bravo'), 'Email*' (filled with 'nacho.bravo@gmail.com'), 'Contraseña*' (filled with '.....'), and 'Repite contraseña*' (filled with '.....'). A blue 'Registrar' button is at the bottom. Below the form, a status message says 'Registrando...' with a circular progress icon. A note at the bottom states: 'Los campos marcados con (*) son obligatorios.'

Fuente: elaboración propia.

La **Figura 34** muestra el formulario de registro de usuarios. Se recopilan los datos necesarios para generar el par de claves y vincular al usuario con su nueva identidad descentralizada.

6.3.4. Autenticación mediante desafío y firma digital

El sistema de autenticación sigue un modelo de desafío/respuesta basado en firma ED25519.

La secuencia completa es la siguiente:

1. El cliente selecciona un usuario y su conjunto de claves desde el navegador.
2. Introduce su contraseña para desbloquear (descifrar) su clave privada.

3. Envía una petición POST /auth/challenge al nodo *backend*, indicando el `user_id`.
4. El *backend* responde con un challenge aleatorio que se asocia con ese `user_id`.
5. El usuario firma el challenge con su clave privada:

```
const challengeBytes = toBytes(challenge);
const signature = await ed.sign(challengeBytes, privateKey);
...
```

6. Envía un POST /auth/verify con los campos { `user_id`, `challenge`, `signature` }.
7. El *backend* verifica la firma y si es válida, responde con un token de sesión temporal y 200 OK.
8. Si la respuesta es positiva, el usuario accede a la vista principal (list.html). Si falla, se le muestra un mensaje de error y se mantiene en la página de *login*.

Figura 35. Cliente web: autenticación de usuario



Fuente: elaboración propia.

La **Figura 35** muestra el formulario de registro. Desde aquí se puede elegir la identidad almacenada en el navegador (en caso de que se hayan generado varias) e introducir la *passphrase* para acceder a la clave privada.

6.3.5. Subida de ficheros

Cuando el usuario sube un fichero:

1. Se cifra localmente mediante el algoritmo AES-GCM.
2. Se genera la estructura metadata con `file_id`, `owner`, `mimetype`, etc.

3. Se envían metadatos y datos cifrados mediante POST multiform a la API:

El código resumido que muestra el envío a través de API es el siguiente:

```
const formData = new FormData();
formData.append('file', new Blob([fileDataEncrypted]), metadata.original_path);
formData.append('metadata', JSON.stringify(metadata));

await fetch('/api/v1/files', {
  method: 'POST',
  headers: { 'Authorization': 'Bearer ' + token },
  body: formData
});
```

Figura 36. Cliente web: subida de ficheros



Fuente: elaboración propia.

La **Figura 36** muestra la interfaz de subida de archivos. El usuario selecciona un fichero y lo cifra localmente antes de enviarlo. Este enfoque garantiza que el contenido esté protegido desde el origen.

6.3.6. Descarga y descifrado

Cuando el usuario descarga un fichero, el cliente web debe ejecutar un conjunto de acciones para validar y descifrar su contenido. Los ficheros se almacenan con cifrado simétrico y cada usuario autorizado recibe una copia de la clave compartida (cifrada para él). El proceso es el siguiente:

1. En la lista de archivos, cada botón de descarga lleva asociado un data-id y data-filename.

La descarga se activa con:

```
$(document).on('click', '.download-btn', async function () {
  const filename = $(this).data('filename');
  const response = await fetch(`/api/v1/files/${filename}`, {
    headers: { Authorization: 'Bearer ' + token }
```

```
});  
  
const blob = await response.blob();  
...  
...
```

2. Cuando el usuario pulsa en el botón o link de descarga, se lanza una petición GET /api/v1/files/<filename> que devuelve el contenido binario cifrado (Blob) junto a la información necesaria para su descifrado en forma de cabeceras (clave compartida, clave pública del propietario que cifró el fichero, vectores de inicialización, etc.) y metadatos (mimetype, size, sha256, ...).
3. El endpoint GET /files/{filename} verifica que el usuario sea válido, esté autenticado y tenga acceso al fichero (comprobando su presencia en el campo authorized_users del fichero de metadatos). Si el nodo que recibe la solicitud dispone de una copia local, se sirve directamente. En caso contrario, actúa como proxy, consultando la lista de replica_nodes y solicitando el archivo a los nodos fuente en paralelo⁵⁰ con el objetivo de optimizar la descarga. Esta se produce en streaming, a la vez que almacena una copia local para usos futuros. A título ilustrativo, se adjunta un extracto del código de paralelización:

```
tasks = [  
    asyncio.create_task(fetch_wrapper(node, file_id))  
    for node in replica_nodes  
]  
  
for task in asyncio.as_completed(tasks):  
    if (stream := await task):  
        # Cancelar las tareas restantes  
        for t in tasks:  
            if t is not task:  
                t.cancel()  
  
        # Actuamos como proxy, guardando una copia local  
        return StreamingResponse(  
            stream_and_store(stream, storage_path, file_id),  
            media_type="application/octet-stream",  
            headers=headers  
        )  
    ...  
...
```

4. De vuelta en el lado cliente, se descifra la clave compartida con:

⁵⁰ Uno de los mayores desafíos del proyecto ha sido conseguir optimizar la descarga de ficheros mediante la consulta en paralelo a múltiples fuentes (replicas).

```
const sharedKey = await decryptSymmetricKey(encryptedKey, iv, privateKey,  
senderPublicKey);  
...
```

5. Con la clave compartida, se descifra el contenido del fichero:

```
const decrypted = await crypto.subtle.decrypt(  
  { name: "AES-GCM", iv: fileIV },  
  aesKey,  
  fileDataEncrypted  
)  
...
```

6. Se presenta el fichero descifrado al usuario final para su descarga y/o visualización.

Figura 37: Cliente web: descarga de fichero

The screenshot shows a web application interface titled 'dfs3 Mis archivos'. At the top right, there are 'Hola, ana' and 'Cerrar sesión' buttons. A blue 'Subir archivo' button is located at the top right. Below the header is a table with columns: Nombre, Tamaño, Fecha de Creación, Acciones, and Tipo. The table contains two rows: '1234.txt' (5 B, 31 may 2025, 14:06:00) and 'test.pdf' (11.8 KB, 31 may 2025, 13:35:50). Each row has four action buttons: 'Descargar' (blue), 'Compartir', 'Renombrar', and 'Borrar'. At the bottom of the page, there is a footer with the text 'José Ignacio Bravo Vicente <nacho.bravo@gmail.com>, UNIR - Universidad Internacional de La Rioja (2025)' and 'Sistema Blockchain de Almacenamiento Distribuido para Dispositivos IoT'.

Fuente: elaboración propia.

La **Figura 37** muestra la vista principal del cliente web. Desde esta interfaz es posible descargar el contenido cifrado y proceder a su descifrado local, bien a través del botón ‘Descargar’ o pinchando directamente en el nombre del fichero.

El nodo nunca conoce la clave compartida ni accede al contenido descifrado. Cada usuario autorizado tiene una copia de la clave compartida solo accesible con su clave privada. El proceso de descifrado ocurre exclusivamente en el lado cliente (navegador). Se mantiene así un enfoque *zero-knowledge*.

6.3.7. Compartición de ficheros

El sistema permite compartir archivos con otros usuarios de la red, sin necesidad de duplicar el contenido y manteniendo confidencialidad, control de acceso y simplicidad en la gestión.

Para ello, se apoya en tres pilares básicos: el cifrado asimétrico de la clave compartida, el registro de usuarios autorizados en los metadatos del fichero compartido y la creación de una entrada visible en el espacio del usuario destino. El proceso resumido es el siguiente:

1. Selección del usuario destino: desde el cliente web, el usuario selecciona el destinatario a partir de una lista. Esta lista se genera con el *endpoint* GET /users e incluye la clave pública de cada usuario.
2. Cifrado de la clave compartida: el archivo ya está cifrado con una clave simétrica aleatoria (AES-256). En el navegador se deriva una clave mediante ECDH (X25519) usando la clave privada del emisor y la clave pública del receptor. Con esta clave, se cifra la clave compartida original, que se guarda junto a su vector de inicialización (IV).
3. Llamada a endpoint: se lanza una solicitud de compartición (POST /files/share) con una estructura parecida a la siguiente:

```
{  
  "filename": "informe_tfg.pdf",  
  "file_id": "a1b2c3...f8f9",  
  "authorized_users": [  
    {  
      "user_id": "e638b76feb3b...",  
      "encrypted_key": "+lGUKfg/wYqsW...",  
      "iv": "WYUjVrJfeGG..."  
    }  
  ]  
}
```

4. En cada nodo, se actualiza el fichero .meta/<file_id>.json, añadiendo al nuevo usuario en la sección authorized_users.
5. Se crea una nueva entrada visible en el directorio raíz del usuario destinatario: .users/<user_id>/<filename>.

Figura 38. Cliente web: compartición de fichero



Fuente: elaboración propia.

La **Figura 38** muestra el cuadro de diálogo de compartición de archivos. El usuario puede seleccionar el destinatario. La operación, permite establecer permisos de acceso seguros sin duplicar contenido.

En la implementación de las entradas virtuales, al usar enlaces sobre un mismo fichero físico **se evita la duplicación de datos** y se mejora la escalabilidad. En lo que respecta a la seguridad, el contenido del archivo nunca se descifra en el nodo. Cada receptor recibe su copia cifrada de la clave compartida que solo podrá descifrar si posee la clave privada para la que se cifró. Es importante recalcar que la compartición no implica modificación del contenido, solo la expansión de los permisos. Cada acción queda registrada mediante eventos `file_shared`.

7. Pruebas y validación

La fase de pruebas es fundamental en cualquier desarrollo de software. Permite detectar errores y validar el comportamiento antes de pasar a producción. Su objetivo es verificar que los componentes cumplen con los requisitos establecidos, tanto de forma individual como integrada y que el sistema en su conjunto responde correctamente ante diferentes escenarios de uso. Se ha estructurado en los siguientes bloques:

- **Pruebas unitarias:** verifican el funcionamiento de funciones específicas como el cifrado/descifrado de claves, generación de identificadores, validación de firmas digitales o lectura de configuración.
- **Pruebas funcionales:** aseguran, entre otras cosas, que cada *endpoint* de la API REST responda según lo esperado. Se comprueba la subida de archivos, la compartición con usuarios o la recuperación de archivos replicados.
- **Pruebas de integración:** evalúan la interacción entre módulos: gestión de eventos, comunicación con IOTA / MQTT, almacenamiento distribuido, etc. Se valida que los eventos se generen y procesen adecuadamente.
- **Pruebas de sistema:** simulan operaciones concurrentes de subida y descarga de archivos, evaluando tiempos de respuesta, consumo de recursos y comportamiento ante situaciones adversas como caída de nodos o pérdidas temporales de conectividad.
- **Pruebas de seguridad:** se valida el cifrado de archivos, la autenticación con firma digital y el control de acceso para asegurar que solo los usuarios autorizados pueden acceder al contenido.

7.1. Pruebas unitarias

La **Tabla 20** recoge una selección de pruebas unitarias realizadas sobre los componentes clave del sistema. Por limitaciones de tiempo, el conjunto de pruebas es limitado, pero lo suficientemente representativo.

Tabla 20. Listado de pruebas unitarias

Función	Prueba Realizada	Resultado Esperado	Resultado
Criptografía	Generación de claves ED25519 / X25519	Claves válidas, formatos correctos.	CORRECTO
Criptografía	Firma y verificación de mensajes	Verificación exitosa con la clave pública correspondiente.	CORRECTO
Criptografía	Cifrado y descifrado de clave compartida con X25519	Contenido recuperado sin alteraciones.	CORRECTO
Criptografía	Derivación de claves con PBKDF2 + salt	Misma entrada y salt generan misma clave.	CORRECTO
Identificadores	Cálculo de user_id y node_id	SHA-256 válido de la clave pública.	CORRECTO
Configuración del nodo	Escritura y lectura de node.json	Persistencia correcta de los campos definidos.	CORRECTO
Configuración del nodo	Descifrado de clave privada con passphrase correcta	Clave restaurada sin errores.	CORRECTO
Eventos	Carga de evento JSON y verificación de firma	Evento válido y firma coincidente.	CORRECTO
Utilidades	Codificación y decodificación base64	Simetría garantizada (encode / decode).	CORRECTO
Utilidades	Formato de timestamp ISO 8601	Fecha generada con precisión temporal correcta.	CORRECTO

Fuente: elaboración propia.

7.2. Pruebas funcionales

Se han diseñado / ejecutado distintos escenarios de prueba a partir de los requisitos funcionales definidos. La **Tabla 21** resume los resultados obtenidos.

Tabla 21. Listado de pruebas funcionales

Caso de Uso	Prueba Realizada	Resultado Esperado	Resultado
CU01 Alta de usuario	Registro nuevo usuario	Usuario registrado y propagado.	CORRECTO
CU02 Autenticación de usuario	Inicio de sesión mediante firma de challenge	Token válido generado tras verificación.	CORRECTO
CU03 Alta de nodo	Registro de nuevo nodo	Nodo registrado y propagado.	CORRECTO
CU04 Actualización de estado de nodo	Envío periódico de estado	Estado actualizado en el resto de nodos.	CORRECTO

Caso de Uso	Prueba Realizada	Resultado Esperado	Resultado
CU05 Subida de fichero	Subida correcta de fichero cifrado	Fichero visible y replicado.	CORRECTO
CU06 Descarga de fichero	Descarga y descifrado de fichero	Fichero recuperado sin errores.	CORRECTO
CU07 Compartición de fichero	Compartir fichero con otro usuario	Acceso permitido y funcional.	CORRECTO
CU08 Listado de ficheros	Consulta de ficheros visibles del usuario	Lista de entradas correctamente mostrada.	CORRECTO
CU09 Renombrado de fichero	Cambio de nombre de una entrada virtual	Entrada renombrada sin afectar al contenido.	CORRECTO
CU10 Borrado de fichero	Eliminación de una entrada virtual	Fichero eliminado del espacio virtual del usuario.	CORRECTO
CU11 Replicación de fichero	Replica automática en otro nodo	Contenido disponible redundante.	CORRECTO

Fuente: elaboración propia.

7.3. Pruebas de integración

La **Tabla 22** resume las pruebas llevadas a cabo para verificar la interacción entre los distintos módulos del sistema. Se han cubierto los flujos más críticos.

Tabla 22. Listado de pruebas de integración

Componentes	Prueba Realizada	Resultado Esperado	Resultado
Cliente web + API REST	Subida de fichero desde navegador	Fichero cifrado, metadatos generados y entrada creada.	CORRECTO
API + Almacenamiento local	Creación de metadatos y guardado en .meta/ y .storage/	Archivos generados correctamente en ambas ubicaciones.	CORRECTO
API + IOTA + MQTT	Emisión de evento file_created tras subida	Evento recibido por otros nodos con metadatos completos.	CORRECTO
Nodo receptor + replicación	Recepción y proceso de evento file_created	Nodo descarga replica y emite file_replicated.	CORRECTO
Cliente web + API + caché SQLite	Solicitud de lista de usuarios para compartir	Usuarios devueltos correctamente desde caché o base de datos.	CORRECTO
API + Validación criptográfica	Verificación de firma en eventos entrantes	Firma válida con clave pública del nodo emisor.	CORRECTO

API + Lógica de compartición	Compartir fichero y crear entrada en el espacio del receptor	Entrada creada y clave compartida cifrada añadida.	CORRECTO
Nodo + recuperación remota	Descarga de réplica desde nodo remoto	Fichero servido correctamente en modo streaming.	CORRECTO

Fuente: elaboración propia.

7.4. Pruebas de sistema

Con el objetivo de evaluar el comportamiento del sistema bajo condiciones de carga y tolerancia a fallos, se han llevado a cabo algunas pruebas de rendimiento centradas en operaciones como la subida, descarga y replicación de archivos. Por otro lado, se han simulado distintos escenarios como la caída de nodos o la pérdida de conectividad. La **Tabla 23** muestra un resumen de los resultados obtenidos.

Tabla 23. Listado de pruebas de rendimiento

Escenario	Prueba Realizada	Resultado Esperado	Resultado	Métricas Obs.
Subida de fichero	Subida sin carga concurrente de fichero de 10 MB	Tiempo esperado ≈ 1-2 s en red local ⁵¹	CORRECTO	Tiempo medio de subida 5248 ms; por debajo de 2 s descontando latencia IOTA
Descarga de fichero	Descarga sin carga concurrente de fichero de 10 MB	Tiempo esperado ≈ 1-2 s en red local	CORRECTO	Tiempo medio de descarga 1572 ms
Subida concurrente de ficheros	10 usuarios suben ficheros simultáneamente	Peticiones aceptadas sin errores ni bloqueos	CORRECTO	100% completadas
Descarga desde múltiples nodos	10 descargas paralelas de 10 MB desde varios nodos	Archivos servidos correctamente con buena latencia	CORRECTO	Tiempo promedio por descarga 8918 ms
Tiempos de respuesta API	Medición sobre endpoints GET /files, ...	Respuesta < 250 ms en operaciones básicas	CORRECTO	GET promedio: 124 ms
Caída de nodo replicador	Apagar nodo que almacena réplica	Archivo sigue siendo accesible desde otros nodos	CORRECTO	El sistema descargó desde nodo alternativo

⁵¹ Límite de velocidad para dispositivos *Fast Ethernet*

Escenario	Prueba Realizada	Resultado Esperado	Resultado	Métricas Obs.
Caída del nodo origen	Nodo que subió el archivo deja de estar disponible	Réplica accesible gracias a nodos secundarios	CORRECTO	replica_nodes contenía suficientes copias
Desconexión temporal de red	Nodo pierde conexión brevemente	Reenvío automático de eventos tras reconexión	CORRECTO	Reenvío por MQTT verificado
Lectura interrumpida	Se corta una descarga mientras está en curso	El cliente recupera el archivo en un nuevo intento	CORRECTO	Descarga reintentada con éxito
Consumo de recursos sostenido	Carga de trabajo prolongada con múltiples operaciones	Sistema operativo estable sin fuga de recursos	CORRECTO	RAM: +45 MB; CPU estable < 50%

Fuente: elaboración propia.

Los tiempos medios observados en las operaciones de subida y descarga de archivos (tanto individual como concurrentemente) están condicionados por varios factores inherentes a la arquitectura del sistema:

- Las limitaciones físicas de los dispositivos IoT empleados (tarjetas *Orange Pi One* con interfaces *Fast Ethernet*), imponen un techo al rendimiento de red.
- La latencia introducida por la red IOTA, utilizada como capa de control y persistencia de eventos, representa una de las principales fuentes de retardo, especialmente en las fases de confirmación y propagación de transacciones.

Por otro lado, las pruebas de concurrencia con 10, 20 e incluso 30 peticiones simultáneas arrojaron resultados sorprendentemente positivos teniendo en cuenta las limitadas capacidades hardware de los nodos utilizados.

Cabe destacar que, durante las pruebas, inicialmente se detectaron **tiempos de descarga por encima de los 20 segundos** para peticiones únicas. Tras analizar el código, se descubrió que el problema estaba en un fallo de diseño:

- Por un lado, el *backend* cargaba en memoria el fichero completo antes de devolverlo. Esto afectaba considerablemente el tiempo de respuesta y aumentaba el consumo de recursos. Además, hacía inviable la descarga de ficheros grandes. Se solucionó haciendo que la descarga fuera en streaming.

- Por otro lado, la latencia de la red IOTA, durante la emisión de un evento (en este caso de auditoría `file_accessed`) puede llegar a ser muy alta, con tiempos que van entre uno y cuatro segundos. Estos eventos se generaban de forma síncrona antes de devolver los datos al navegador, lo que, de nuevo retrasaba la respuesta inicial e incrementaba la duración de la operación. Se solucionó haciendo que el evento se envíe de forma asíncrona y paralela una vez iniciado el proceso de descarga.

Una vez refactorizado se consiguieron tiempos mucho más razonables (por debajo de los dos segundos). El sistema fue capaz de gestionar todas las solicitudes paralelas sin provocar bloqueos, caídas de rendimiento graves ni un consumo excesivo de recursos. El uso de operaciones asíncronas y en streaming permitió mantener una carga de CPU moderada con una huella de memoria estable.

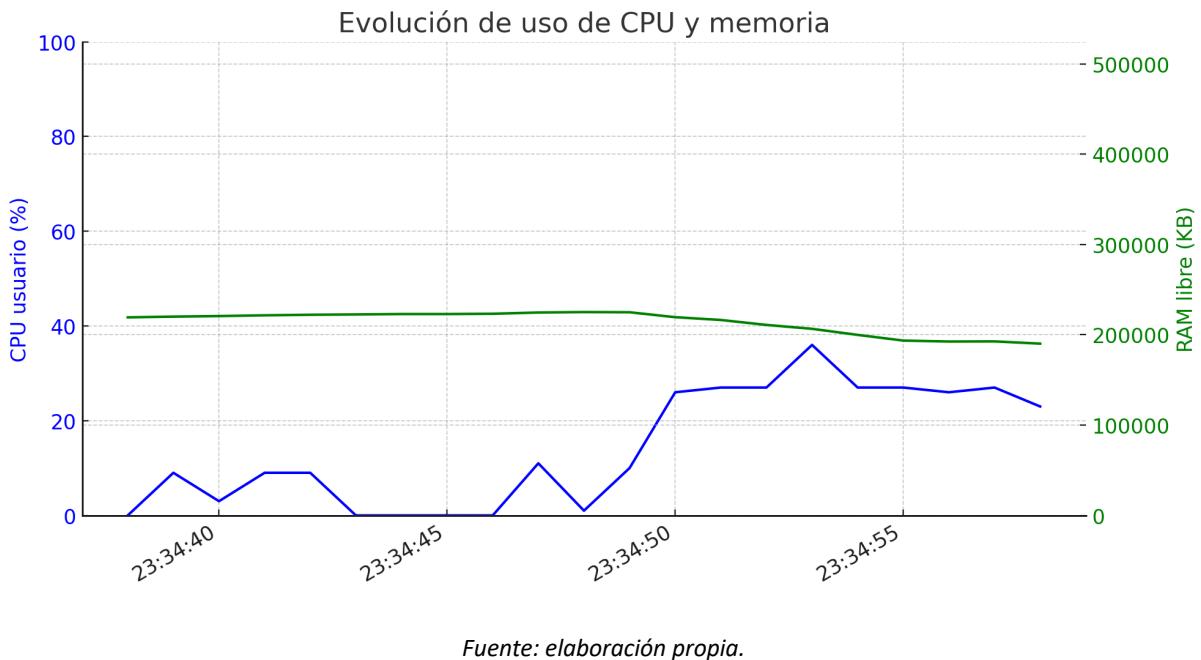
Tabla 24. Comportamiento del sistema ante cargas concurrentes

Métrica	5 procesos	10 procesos	20 procesos
r (run queue)	1	1–2	hasta 3
cs (context switches)	~1400–1700	~1600–2300	hasta 3000+
%us (user CPU)	3–8%	4–13%	hasta 13%
%sy (kernel CPU)	0–1%	1–2%	hasta 2%
%id (idle CPU)	91–97%	85–96%	baja hasta 86%
%wa (I/O wait)	0%	0%	0%
b (blocked procs)	0	0	sigue en 0
free (RAM disponible)	estable (~232k)	estable (~225k)	baja un poco (~208k)
bo (blocks out)	leve	leve	hasta 67
in/cs pico (interrupts)	~300–500	~500–1300	hasta 2500–3000

Fuente: elaboración propia.

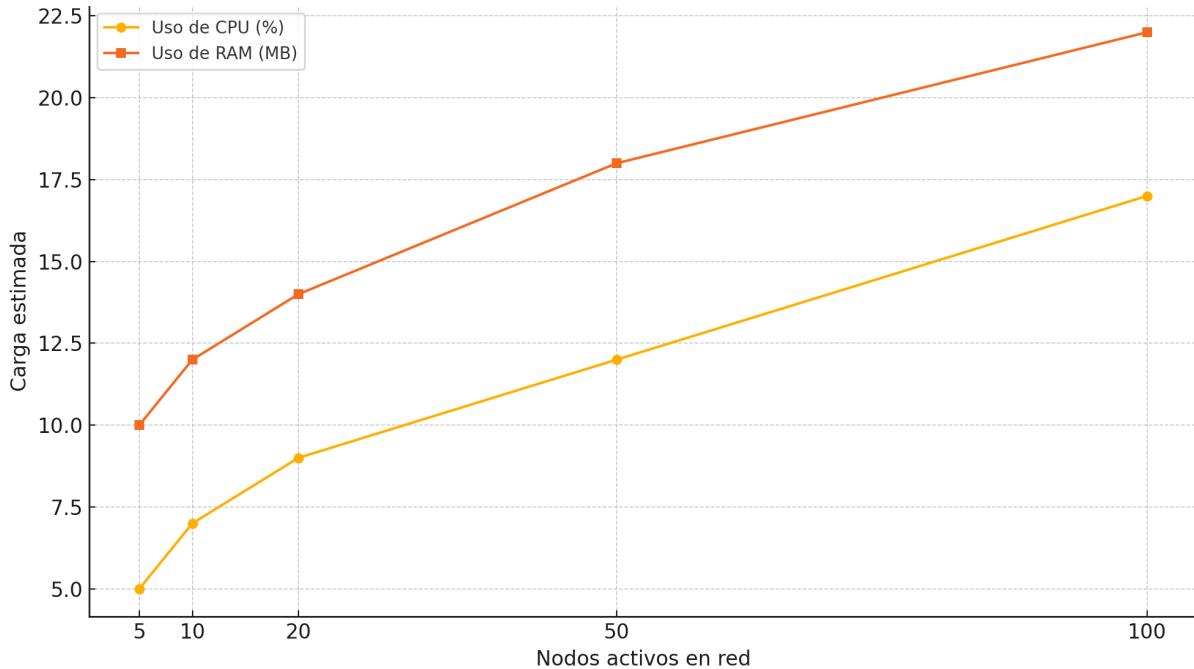
La **Tabla 24** muestra el comportamiento del sistema ante cargas concurrentes con métricas observadas para 5, 10 y 20 procesos simultáneos. La **Figura 39** muestra la evolución en el uso de CPU y memoria durante un pico de actividad de 20 peticiones concurrentes. Observamos que el uso de CPU no llega a superar el 40% de utilización y la caída de memoria libre está por debajo de los 30 MB.

Figura 39. Evolución de uso de CPU y memoria durante 20 peticiones concurrentes



Fuente: elaboración propia.

Figura 40. Carga estimada de CPU y memoria por número de nodos activos



Fuente: elaboración propia.

La **Figura 40** muestra la carga estimada de CPU y uso de memoria por nodo en función del número de nodos activos en la red. La simulación parte de un escenario de actividad distribuida, donde cada nodo genera eventos periódicos (`node_status`) y esporádicamente

realiza accesos a archivos remotos (`file_accessed`) o participa en operaciones como la creación y compartición de ficheros (`file_created`, `file_shared`), replicación (`file_replicated`), eliminación de entradas virtuales (`file_deleted`) o gestión de usuarios (`user_created`, `user_joined_node`). Los valores reflejan una carga media esperada, sin picos extremos de concurrencia⁵².

7.5. Pruebas de seguridad

Se han validado los mecanismos criptográficos usados en el cifrado de archivos, la compartición segura de claves simétricas y la autenticación por firma digital. También se han simulado intentos de acceso no autorizado y de manipulación de eventos para comprobar la robustez del sistema frente a ataques. La **Tabla 25** muestra los resultados obtenidos.

Tabla 25. Listado de pruebas de seguridad

Aspecto Evaluado	Prueba Realizada	Resultado Esperado	Resultado	Observaciones
Cifrado de archivos	Subida de archivo cifrado y verificación del contenido local	Contenido ilegible sin la clave correspondiente	CORRECTO	Confirmado en .storage/
Cifrado de claves simétricas	Compartir archivo y verificar clave cifrada por receptor	Solo el usuario autorizado puede descifrar la clave	CORRECTO	Verificado con claves cruzadas
Verificación de firmas	Validación de evento con firma alterada	Firma rechazada	CORRECTO	Evento descartado por inconsistencia
Acceso no autorizado	Intentar descargar archivo sin permisos	Acceso denegado	CORRECTO	Código HTTP 403 devuelto
Reutilización de token de sesión	Usar token caducado o inválido	Rechazo de la petición	CORRECTO	Sesión invalidada correctamente

⁵² Carga estimada de CPU y uso de memoria en un nodo en función del número de nodos activos en la red. Esta proyección se basa en una simulación donde cada nodo genera entre 75 y 100 eventos por hora, distribuidos aproximadamente en un 70 % de eventos `node_status`, un 20 % de eventos `file_accessed` y el 10 % restante repartido entre operaciones `file_created`, `file_shared`, `file_deleted`, `file_replicated`, `file_renamed`, `user_created` y `user_joined_node`. Las estimaciones se apoyan en mediciones empíricas de carga bajo condiciones de envío masivo de eventos. Reflejan una situación de actividad media sostenida, sin concurrencia extrema. Se considera que cada nodo actúa tanto como origen, como destino, distribuyendo la carga de forma uniforme.

Aspecto Evaluado	Prueba Realizada	Resultado Esperado	Resultado	Observaciones
Confidencialidad en transporte	Intercepción de tráfico entre cliente y nodo (MITM simulado)	Imposibilidad de leer datos cifrados	CORRECTO	Contenido cifrado desde el cliente
Integridad de evento	Modificar manualmente campo de un evento recibido	Firma no válida, evento ignorado	CORRECTO	Firma actúa como mecanismo de integridad

Fuente: elaboración propia.

7.6. Resumen de validación de requisitos

A partir de los resultados, se concluye que el sistema cumple con los requisitos establecidos en la fase de análisis. Se han validado sus funcionalidades, confirmando su capacidad de gestionar usuarios, nodos y ficheros de forma segura, descentralizada y redundante a pequeña escala. Las pruebas de rendimiento y tolerancia a fallos han demostrado que es la solución es capaz de mantener una buena capacidad de resistencia ante situaciones adversas.

Aunque algunas pruebas han quedado fuera del alcance por limitaciones de tiempo, creemos que el grado de cobertura alcanzado permite considerar el sistema lo suficientemente maduro, estable y funcional en su versión actual.

8. Conclusiones y trabajo futuro

A continuación, se recogen las principales conclusiones y se proponen varias líneas de mejora que pueden servir de base para futuros trabajos.

8.1. Conclusiones del trabajo

El objetivo del proyecto ha sido diseñar e implementar un sistema de almacenamiento de ficheros distribuido, descentralizado y seguro, basado en dispositivos IoT de bajo consumo. Para conseguirlo, **se han desarrollado los componentes necesarios y se han integrado en un hardware específico**. La fase de pruebas ha permitido **validar su funcionalidad**, rendimiento, seguridad y tolerancia a fallos.

Como primer punto, destaca la **naturaleza híbrida del proyecto**, donde han convergido aspectos de desarrollo de software con consideraciones propias de la integración de sistemas embebidos que han requerido de un enfoque integral. Uno de los grandes retos ha consistido en la **implementación de la capa criptográfica** (tanto en el lado cliente como en nodo): la gestión de claves, el intercambio seguro de información en entornos distribuido, la validación de integridad, ... todo ello ha requerido de un entendimiento teórico avanzado con múltiples iteraciones de prueba y error hasta cumplir con los objetivos. El sistema de descarga en paralelo para **maximizar el rendimiento y minimizar la latencia** en redes distribuidas también ha supuesto un desafío importante, al igual que la necesidad de **ejecución con un consumo mínimo de recursos**. La **integración con IOTA**, una tecnología todavía en evolución, ha sido problemática: la falta de estabilidad en sus herramientas (ej. la obsolescencia de PyOTA o la incompletitud de las nuevas librerías), una documentación desactualizada y los cambios frecuentes en su ecosistema de pruebas, han dificultado considerablemente el desarrollo. IOTA ofrece un enfoque interesante para entornos distribuidos, pero **su falta de madurez genera reservas en términos prácticos**.

Si bien los resultados demuestran la viabilidad del sistema, **se han revelado ciertas limitaciones en su rendimiento**, especialmente en lo que respecta a las operaciones de subida y descarga de ficheros. Estos cuellos de botella se deben tanto a las restricciones propias de los dispositivos utilizados (con procesadores modestos y ancho de banda limitado, algo que era de esperar), como a la latencia de la *blockchain* empleada (*Tangle* de IOTA). Pese a todo,

las pruebas han demostrado que **el sistema es capaz de mantener un comportamiento coherente y estable** ante cargas de trabajo más o menos moderadas.

Se han identificado varios **riesgos**. La posible **pérdida de claves privadas** por parte del usuario se mitiga exportando una copia protegida con contraseña. La **caída o desconexión de nodos** se aborda replicando los ficheros en múltiples nodos para garantizar su disponibilidad. Los **problemas de latencia** asociados a la Tangle de IOTA se complementan con un canal MQTT con tiempos de respuesta bajos y acciones asíncronas. Los **cuellos de botella** por la capacidad limitada de cómputo de los dispositivos IoT, se reducen gracias a su implementación ligera y modular. Se ha considerado también el riesgo derivado de la **evolución o desactualización de librerías externas** (como el SDK de IOTA), para lo cual se ha optado por una arquitectura desacoplada que permite la sustitución de módulos sin impacto global... Aunque estas medidas no eliminan el riesgo por completo, proporcionan un marco sólido que ayuda a mitigarlo.

Por último, mencionar la **alineación de los resultados obtenidos con los principios fundamentales del paradigma Web 3.0**. Es particularmente interesante el modelo de **identidad digital descentralizada**, basado en claves criptográficas, generadas localmente por el usuario sin necesidad de aportar información personal. Refuerza la soberanía individual y elimina la dependencia de entidades centrales para garantizar la autorización y la autenticación. Esta aproximación puede representar un avance hacia nuevos sistemas, donde el usuario sea propietario de su identidad en el ecosistema digital.

Como conclusión, aunque se han simplificado varias funcionalidades por limitaciones de alcance y pese a ciertos compromisos en términos de rendimiento, creemos que se han cumplido los objetivos y que los resultados demuestran la viabilidad técnica de la idea inicial, pudiendo esta servir de base para futuros trabajos.

8.2. Líneas de trabajo futuro

Como continuación del proyecto, se proponen varias líneas de mejora. Algunas ya se han analizado y aparecen documentadas en el apartado anexos:

1. Evolucionar el sistema de replicación actual a **un modelo más robusto y eficiente**, reforzando la lógica de reintentos, verificación de estado y control de errores. La

integración de un sistema de autenticación basado en tokens JWT, facilitaría su uso distribuido (actualmente no hay sincronización de sesiones entre nodos). Otras propuestas, incluyen una interfaz gráfica mejorada con capacidades de monitorización de nodos, eliminación segura de datos y/o control de versiones.

2. Estudiar su **escalabilidad** en entornos con decenas, cientos e incluso miles de nodos, mediante aplicación de técnicas como *sharding*⁵³. Se podrían integrar mecanismos de reputación para priorizar a los nodos que colaboran más activamente.
3. Migrar a un **lenguaje de programación más eficiente** como *Rust*, *Go* o incluso *C/C++*. Aunque Python ha facilitado el desarrollo rápido del prototipo, su consumo de memoria y rendimiento en dispositivos embebidos puede suponer una limitación para equipos con recursos hardware limitados.
4. Aprovechar su arquitectura descentralizada para tareas como la **ejecución distribuida de código**, ampliando su funcionalidad más allá del almacenamiento de datos. Esta evolución permitiría a los nodos ejecutar funciones o scripts de forma segura, abriendo la puerta a nuevos escenarios como el procesamiento local, la automatización de tareas o el despliegue de lógica personalizada en forma de funciones *serverless* distribuidas. El enfoque estaría alineado con los actuales **principios de computación en la niebla** o *fog computing*.
5. Finalmente, la **incorporación de incentivos basados en contratos inteligentes dentro de la red IOTA** supondría una evolución hacia un modelo completamente descentralizado y económicamente sostenible. El usuario pagaría una cantidad justa por el uso de los servicios distribuidos ofrecidos por “micro-proveedores”, que recibirían a cambio una compensación (aportarían su hardware, comunicaciones y electricidad). Esta funcionalidad sería especialmente relevante en **escenarios de colaboración abierta** con escalabilidad de ámbito global.

⁵³ El *Sharding* es una técnica de particionado que consiste en dividir una base de datos o red en fragmentos más pequeños llamados *shards* que se distribuyen entre diferentes nodos o servidores.

Referencias bibliográficas

- Ali, S., Wang, G., White, B., & Cottrell, R. L. (2018). *A Blockchain-Based Decentralized Data Storage and Access Framework for PingER*. Obtenido de <https://www.osti.gov/servlets/purl/1475405>
- Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., & Yellick, J. (2018). *Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains*. Obtenido de Proceedings of the Thirteenth EuroSys Conference, 30, 1–15: <https://doi.org/10.1145/3190508.3190538>
- Anwar, H. (2018). *Web 3.0 Blockchain Technology Stack: The Comprehensive Guide*. Obtenido de 101 Blockchains: <https://101blockchains.com/web-3-0-blockchain-technology-stack/>
- Bapatla, A. K., Deepak, P., Mohanty, S. P., Yanambaka, V. P., & Kougianos, E. (2023). *EasyChain: an IoT-friendly blockchain for robust and energy-efficient authentication*. Obtenido de Frontiers: <https://doi.org/10.3389/fbloc.2023.1194883>
- Bapatla, A. K., Mohanty, S. P., & Kougianos, E. (2022). *PharmaChain: Blockchain-based smart contract platform for pharmaceutical supply chain management*. Obtenido de arXiv: <https://arxiv.org/abs/2202.02592>
- Benet, J. (2014). *IPFS - Content Addressed, Versioned, P2P File System*. Obtenido de <https://ipfs.io/ipfs>
- Bocek, T., Rodrigues, B. B., Strasser, T., & Stiller, B. (2017). *Blockchains Everywhere - A Use-case of Blockchains in the Pharma Supply-Chain*. Obtenido de IFIP Open Digital Library: <https://dl.ifip.org/db/conf/im/im2017exp/119.pdf>
- Boyce, S. (2019). *Yes, You Can Put IoT On The Blockchain Using Python And The ESP8266*. Obtenido de Hackaday: <https://hackaday.com/2019/03/01/yes-you-can-put-iot-on-the-blockchain-using-python-and-the-esp8266>
- Buterin, V. (2014). *A Next-Generation Smart Contract and Decentralized Application Platform*. Obtenido de Ethereum White Paper: <https://ethereum.org/en/whitepaper/>

- Campbell, C. (2025). *What are the 4 different types of blockchain technology?* Obtenido de TechTarget: <https://www.techtarget.com/searchcio/feature/What-are-the-4-different-types-of-blockchain-technology>
- Campbell, R. (2016). *Modum.io's Tempurature-Tracking Blockchain Solution Wins Accolades at Kickstarter Accelerator 2016.* Obtenido de Bitcoin Magazine: <https://bitcoinmagazine.com/business/modum-io-s-tempurature-tracking-blockchain-solution-wins-accolades-at-kickstarter-accelerator-1479162773>
- Cardano Spot. (s.f.). *First Static Website on Cardano.* Obtenido de <https://cardanospot.io/news/first-static-website-on-cardano-1>
- Chen, X., He, C., Chen, Y., & Xie , Z. (2023). *Internet of Things (IoT)—blockchain-enabled pharmaceutical supply chain resilience in the post-pandemic era.* Obtenido de Front. Eng. Manag. 10: <https://doi.org/10.1007/s42524-022-0233-1>
- Dave. (6 de 9 de 2023). X. Obtenido de https://x.com/ItsDave_ADA/status/1699442740050702464
- DrugPatentWatch. (2024). *Real World Blockchain Uses in the Pharmaceutical Industry.* Obtenido de DrugPatentWatch: <https://www.drugpatentwatch.com/blog/real-world-blockchain-uses-in-the-pharmaceutical-industry>
- El Hakim, A. (marzo de 2018). *Internet of Things (IoT) System Architecture and Technologies.* Obtenido de ResearchGate: <http://dx.doi.org/10.13140/RG.2.2.17046.19521>
- Froiz-Míguez, I., Fraga-Lamas, P., Varela-Barbeito, J., & Fernández-Caramés, T. M. (2019). *LoRaWAN and Blockchain based Safety and Health Monitoring System for Industry 4.0 Operators.* Obtenido de MDPI: <http://dx.doi.org/10.3390/ecsa-6-06577>
- Ghadge, A., Bourlakis, M., Kamble, S., & Seuring, S. (2022). *Blockchain implementation in pharmaceutical supply chains: A review and conceptual framework.* Obtenido de International Journal of Production Research: <https://doi.org/10.1080/00207543.2022.2125595>
- Hamilton, D. (2024). *Top 10 IoT Blockchain Projects that Have the Potential to Shake Up The Market.* Obtenido de Securities.io: <https://www.securities.io/top-iot-blockchain-projects>

Hoskinson, C. (2017). *Cardano and the Ouroboros Protocol*. Obtenido de IOHK:
<https://iohk.io/en/research/library>

IBM. (2025). *Benefits of blockchain* . Obtenido de IBM:
<https://www.ibm.com/think/topics/benefits-of-blockchain>

IBM. (2025). *¿Qué es el blockchain?* Obtenido de IBM: <https://www.ibm.com/es-es/topics/blockchain>

Innovative Health Initiative. (2025). Obtenido de Innovative Health Initiative:
<https://www.ihi.europa.eu/>

Khalil, R. A., Saeed, N., Fard, Y. M., Al-Naffouri, T. Y., & Alouini, M.-S. (2020). *Deep Learning in Industrial Internet of Things: Potentials, Challenges, and Emerging Applications*. Obtenido de arXiv: <https://arxiv.org/abs/2008.06701>

Krause, D. (2023). *Web3 and the Decentralized Future: Exploring Data Ownership, Privacy, and Blockchain Infrastructure*. Obtenido de SSRN Electronic Journal:
https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5064483

Maymounkov, P., & Mazières, D. (2002). Kademlia: A Peer-to-peer Information System Based on the XOR Metric. *International Workshop on Peer-to-Peer Systems* (págs. 53–65). Springer. Obtenido de <https://www.scs.stanford.edu/~dm/home/papers/kpos.pdf>

Mohammed Abdul, S. S. (2024). *Navigating Blockchain's Twin Challenges: Scalability and Regulatory Compliance*. Obtenido de MDPI:
<https://doi.org/10.3390/blockchains2030013>

Nakamoto, S. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*. Obtenido de <https://bitcoin.org/bitcoin.pdf>

Nano. (2025). *Digital money shouldn't cost the Earth*. Obtenido de Nano:
<https://nano.org/en/sustainability>

Pilkington, E., & Aratani , L. (19 de julio de 2024). *This article is more than 8 months old US transportation, police and hospital systems stricken by global CrowdStrike IT outage*. Obtenido de The Guardian:
<https://www.theguardian.com/technology/article/2024/jul/19/crowdstrike-microsoft-outage>

Protocol Labs. (2020). *Filecoin: A Decentralized Storage Network*. Obtenido de

<https://filecoin.io/>

Rodriguez, N. (12 de septiembre de 2018). *La Web 3.0 estará impulsada por la tecnología*

Blockchain Stack. Obtenido de 101 Blockchains: <https://101blockchains.com/es/web-3-0-tecnologia-blockchain-stack/>

Salih, K. O., Rashid, T. A., Radovanovi, D., & Bacanin, N. (2022). *A Comprehensive Survey on the Internet of Things with the Industrial Marketplace*. Obtenido de Sensors:

<https://www.mdpi.com/1424-8220/22/3/730>

Spiceworks. (2023). *Peer-to-Peer Networks: Features, Pros, and Cons*. Obtenido de

<https://www.spiceworks.com/tech/networking/articles/what-is-peer-to-peer/>

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., & Balakrishna, H. (2001). Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *ACM SIGCOMM Computer Communication Review*, 31(4), 149–160. Obtenido de

<https://doi.org/10.1145/964723.383071>

Storj Labs. (2021). *Whitepaper v3*. Obtenido de <https://www.storj.io/>

Turki, M., Cheikhrouhou, S., Dammak, B., Baklouti, M., Mars, R., & Dhahbi, A. (2022). *NFT-IoT Pharma Chain : IoT Drug traceability system based on Blockchain and Non Fungible Tokens (NFTs)*. Obtenido de Journal of King Saud University – Computer and

Information Sciences:

<https://www.sciencedirect.com/science/article/pii/S1319157822004347/pdf>

Vorick, D., & Champine, L. (2014). *Sia: Simple Decentralized Storage*. Obtenido de

<https://sia.tech>

Whittaker , M. (2024). *The Beginning of the End of Big Tech*. Obtenido de Wired:

<https://www.wired.com/story/the-beginning-of-the-end-of-big-tech>

Wilkinson, S., Boshevski, T., Brandoff, J., & Prestwich, C. (2014). *Storj: A Peer-to-Peer Cloud Storage Network*. Obtenido de <https://storj.io>

Wood, G. (2020). *Polkadot: Vision for a Heterogeneous Multi-Chain Framework*. Obtenido de Web3 Foundation: <https://polkadot.network/Polkadot-whitepaper.pdf>

Yakovenko, A. (2020). *Solana: A new architecture for a high performance blockchain.*

Obtenido de Solana White Paper: <https://solana.com/solana-whitepaper.pdf>

Yousefpou, A., Fung, C., Nguyen , T., Kadiyala, K., Jalali, F., Niakanlahiji, A., . . . Jue, J. P.

(2018). All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98, 289-330.

Obtenido de Science Direct: <https://doi.org/10.1016/j.sysarc.2019.02.009>

Zhang, Y., & Jacobsen, H.-A. (2018). Towards dependable, scalable, and pervasive distributed ledgers with blockchains. *Future Generation Computer Systems*, 107, 819–830.

Obtenido de <https://doi.org/10.1016/j.future.2017.08.048>

Anexo A. Mecanismos de consenso blockchain

Los mecanismos de consenso permiten a los nodos de una *blockchain* acordar el estado de la red sin necesidad de una entidad central. Son esenciales para validar transacciones, mantener la integridad de los datos y garantizar la seguridad del sistema. A continuación, los mecanismos más habituales:

- **PoW (Proof of Work)**: los nodos compiten resolviendo problemas criptográficos. Es seguro, pero altamente consumidor de energía (ej. *Bitcoin*).
- **PoS (Proof of Stake)**: los validadores son seleccionados en función de la cantidad de tokens que poseen y que bloquean en garantía (*stake*⁵⁴). Se reduce el consumo energético (ej. *Ethereum*, *Cardano*).
- **DPoS (Delegated Proof of Stake)**: los usuarios votan por un número limitado de nodos delegados que se encargan de validar bloques. Mejoran la eficiencia, pero reducen la descentralización (ej. EOS, TRON).
- **LPoS (Leased Proof of Stake)**: similar al PoS, pero los usuarios pueden “alquilar” su *stake* a otros nodos sin transferir la propiedad. Se democratiza la participación (ej. *Waves*).
- **PoA (Proof of Authority)**: un conjunto limitado de nodos autorizados valida los bloques, confiando en su reputación. Muy eficiente, pero con menor descentralización (ej. *VeChain* y redes privadas).
- **PoB (Proof of Burn)**: los participantes “queman” (eliminan) tokens para obtener el derecho a validar bloques. Promueve el compromiso a largo plazo (ej. *Slimcoin*).
- **PoET (Proof of Elapsed Time)**: los nodos esperan un período aleatorio seguro antes de proponer un bloque. Determinado por hardware de confianza. Usado en redes privadas (ej. *Hyperledger Sawtooth*).

⁵⁴ Stake o cantidad de criptomonedas que un usuario bloquea o deposita como garantía para participar en el proceso de validación de bloques en redes con mecanismos de consenso como *Proof of Stake*. A mayor *stake*, mayor probabilidad de ser seleccionado como validador y obtener recompensas.

- **PoH (Proof of History)**: permite ordenar transacciones antes de que lleguen al consenso mediante una secuencia criptográfica de marcas de tiempo. Altamente eficiente (ej. *Solana*).
- **PBFT (Practical Byzantine Fault Tolerance)**: los nodos validan bloques por votación. Se toleran fallos o comportamientos maliciosos en hasta 1/3 de la red. Muy eficiente en entornos controlados (ej. *Hyperledger Fabric*).
- **Tendermint BFT**: evolución de PBFT que combina *staking* y consenso tolerante a fallos bizantinos. Usado en arquitecturas multicadena (ej. *Cosmos*).
- **Proof of Space / Capacity**: los usuarios aportan espacio de almacenamiento libre como recurso de consenso en lugar de potencia de cómputo (ej. *Chia*).
- **Hybrid PoW / PoS**: combinación de mecanismos de minería y participación basada en *stake*. Buscan equilibrio entre seguridad y eficiencia (ej. *Decred*).
- **Avalanche Consensus**: basado en submuestreo y decisión probabilística. Alto rendimiento y rápida finalización de transacciones (ej. *Avalanche*).

Anexo B. Tipos de redes blockchain

Existen distintos tipos de redes *blockchain* según su grado de apertura y control:

- **Blockchain pública:** totalmente abierta, permite que cualquier usuario participe en la red sin permisos previos. Ejemplos destacados son *Bitcoin* y *Ethereum*. Ofrece la máxima descentralización y transparencia, aunque con menor rendimiento.
- **Blockchain privada:** gestionada por una entidad u organización específica, solo usuarios autorizados pueden acceder y validar transacciones. Mejora la eficiencia y el control, sacrificando descentralización.
- **Blockchain híbrida:** combina características de las redes públicas y privadas con el objetivo de aprovechar las ventajas de ambas. En este modelo, ciertos datos o procesos se gestionan de forma privada dentro de una organización o consorcio, mientras que otros se registran en una *blockchain* pública para asegurar transparencia, trazabilidad o verificación externa.
- **Blockchain permissionada:** restringe ciertos roles o funciones a nodos con permisos específicos. Puede ser pública o privada y busca un equilibrio entre transparencia, control y rendimiento. Son comunes en entornos empresariales o consorciados.

Anexo C. Principales Redes blockchain

A continuación, se presentan las principales plataformas *blockchain*, destacando sus características, usos y diferencias clave.

- **Bitcoin** (BTC): lanzada en 2009 por Satoshi Nakamoto, es la primera *blockchain* funcional y fue diseñada como un sistema de efectivo electrónico descentralizado. Utiliza un mecanismo de consenso basado en *Proof of Work* (PoW), lo que garantiza alta seguridad a costa de una baja escalabilidad (~7 TPS) y un elevado consumo energético. Su uso se centra en la reserva de valor y transferencias sin intermediarios (Nakamoto, 2008).
- **Ethereum** (ETH): introducida en 2015 por Vitalik Buterin, Ethereum extendió el concepto de *blockchain* al permitir contratos inteligentes y aplicaciones descentralizadas (*dApps*). Migró a POS en 2022 para mejorar su sostenibilidad. Es la red más adoptada para desarrollos en sectores como NFT, *DeFi* (*Decentralized Finance*) y DAO (*Decentralized Autonomous Organizations*) (Buterin, 2014).
- **Cardano** (ADA): desarrollada por Charles Hoskinson en 2017, adopta un enfoque académico, basado en revisiones por pares. Utiliza *Ouroboros*, un algoritmo de PoS formalmente verificado. Ofrece alta eficiencia energética, modularidad y foco en identidad digital / trazabilidad, especialmente en regiones con infraestructuras digitales limitadas (Hoskinson, 2017).
- **Polkadot** (DOT): creada por Gavin Wood (cofundador de Ethereum) en 2020, busca resolver la interoperabilidad entre *blockchains*. Utiliza un modelo multicadena con una *relay chain* y múltiples *parachains*, con consenso *NPoS* y BFT. Está diseñada para facilitar aplicaciones especializadas que requieran cooperación entre cadenas independientes (Wood, 2020).
- **Solana** (SOL): también lanzada en 2020, prioriza el rendimiento a través de un mecanismo híbrido PoH (*Proof of History*) + PoS. Ofrece teóricamente más de 50,000 TPS con bajas comisiones, aunque ha enfrentado críticas por interrupciones y cierta centralización (Yakovenko, 2020). Es popular en sectores como juegos Web3, NFT y *DeFi*.

- **Hyperledger Fabric:** proyecto empresarial iniciado por IBM y gestionado por la *Linux Foundation* desde 2015. Está orientado a *blockchains* permisionadas, modulares y configurables, sin token nativo. Se utiliza principalmente en soluciones B2B para trazabilidad, cumplimiento y automatización de procesos (Androulaki et al., 2018).

Anexo D. Desafíos del almacenamiento distribuido blockchain

Se describen los principales retos identificados en un sistema de almacenamiento distribuido basado en tecnologías *blockchain*:

- **Integridad y verificación de datos off-chain:** la mayoría de las soluciones basadas en *blockchain* almacenan solo referencias o metadatos en cadena, mientras que los datos reales se alojan fuera. Garantizar la integridad y la trazabilidad de estos datos externos sigue siendo un reto, especialmente cuando los nodos no son confiables.
- **Limitaciones de escalabilidad:** las redes *blockchain* tradicionales enfrentan limitaciones en cuanto al número de transacciones por segundo, latencia y consumo de recursos. Aunque existen soluciones como DAG o *sidechains*⁵⁵, muchas aún están en fases tempranas y presentan compromisos en términos de seguridad o descentralización.
- **Gestión de identidad y control de acceso descentralizado:** la ausencia de entidades centrales plantea desafíos en la autenticación, autorización y gestión de identidades digitales en sistemas distribuidos. Es necesario encontrar mecanismos seguros, escalables y respetuosos con la privacidad.
- **Resiliencia frente a nodos poco confiables:** en entornos heterogéneos, como redes de dispositivos IoT, es común que existan nodos intermitentes o con capacidad limitada. Garantizar la disponibilidad del sistema y la replicación de datos en presencia de estos fallos es un desafío técnico.
- **Privacidad y confidencialidad en entornos distribuidos:** asegurar la privacidad de los datos en sistemas P2P es complejo, especialmente cuando los datos se almacenan y replican entre múltiples nodos. El uso de cifrado es esencial, pero plantea retos adicionales en términos de acceso selectivo y control de versiones.
- **Rendimiento y consumo energético:** algoritmos como *Proof-of-Work* han sido cuestionados por su elevado consumo energético. Aunque han surgido alternativas

⁵⁵ *Sidechain* es una cadena de bloques paralela e interoperable con una *blockchain* principal que permite transferir activos entre ambas redes, ofreciendo escalabilidad, privacidad o funcionalidades específicas sin sobrecargar la red principal.

como *Proof-of-Stake* o PBFT, su aplicabilidad en dispositivos IoT de bajo consumo sigue siendo limitada o poco explorada.

Anexo E. Casos de uso extendidos

Se presentan los casos de uso extendidos que detallan en el flujo de acciones, actores involucrados y condiciones específicas para las principales operaciones.

Los casos de uso presentados a continuación fueron diseñados en una etapa temprana con el objetivo de definir la arquitectura general del sistema y pueden no reflejar con exactitud la implementación final. Algunos flujos se han ido adaptado y/o simplificado posteriormente, sobre todo a lo largo de la fase de desarrollo.

Código UML

Se muestra el código UML correspondiente a los casos de uso extendidos. Este código describe de forma estructurada las interacciones entre los actores y las funcionalidades del sistema. Se incluyen extensiones y condiciones que complementan los principales flujos de uso.

```
@startuml

actor Usuario
actor Nodo

Usuario --> (Alta de usuario)
Usuario --> (Autenticacion de usuario)
Usuario --> (Subida de fichero)
Usuario --> (Descarga de fichero)
Usuario --> (Comparticion de fichero)
Usuario --> (Listado de ficheros)
Usuario --> (Renombrado de fichero virtual)
Usuario --> (Borrado de fichero virtual)

Nodo --> (Alta de nodo)
Nodo --> (Actualizacion de estado de nodo)
Nodo --> (Replicacion de fichero en otros nodos)
Nodo --> (Procesar alta de usuario)
Nodo --> (Procesar alta de nodo)
Nodo --> (Procesar replicacion de ficheros)
Nodo --> (Procesar subida de fichero)
Nodo --> (Procesar particion de fichero)
Nodo --> (Procesar descarga de fichero)
Nodo --> (Procesar renombrado de fichero)
Nodo --> (Procesar borrado de fichero)

(Alta de usuario) --> (Procesar alta de usuario) : <<include>>
(Alta de nodo) --> (Procesar alta de nodo) : <<include>>
(Subida de fichero) --> (Procesar subida de fichero) : <<include>>
(Comparticion de fichero) --> (Procesar particion de fichero) : <<include>>
(Descarga de fichero) --> (Procesar descarga de fichero) : <<include>>
```

```
(Renombrado de fichero virtual) --> (Procesar renombrado de fichero) : <<include>>
(Borrado de fichero virtual) --> (Procesar borrado de fichero) : <<include>>
(Replicacion de fichero en otros nodos) --> (Procesar replicacion de ficheros) :
<<include>>
```

@enduml

CU01: Alta de usuario

La **Tabla 26** presenta el caso de uso extendido correspondiente al alta de usuario, detallando los pasos, actores y condiciones asociadas a su ejecución.

Tabla 26. CU01 - Alta de usuario

CU01		Alta de usuario
Actor principal	Usuario (interacción a través de interfaz web en un nodo)	
Resumen	<p>Cuando un usuario accede por primera vez al sistema a través de la web de un nodo, si no dispone de certificado, se le ofrecerá registrarse. El cliente web generará un par de claves derivado de una <i>passphrase</i> y enviará la información necesaria para su alta al nodo. El nodo validará localmente, registrará al usuario en su base de datos, publicará el evento en IOTA, y notificará al resto de nodos vía MQTT.</p>	
Flujo principal	<ol style="list-style-type: none"> 1. El usuario accede al sistema web de un nodo. 2. El sistema detecta que no dispone de certificado válido o que no está registrado y ofrece al usuario la opción de darse de alta. 3. El cliente web: <ul style="list-style-type: none"> • Sigue una <i>passphrase</i> segura al usuario. • Deriva un nuevo par de claves: <ul style="list-style-type: none"> ◦ Clave privada: almacenada localmente en navegador, cifrada. ◦ Clave pública: para verificación y operaciones futuras. • Calcula el <i>user_id</i> como SHA-256 de la clave pública. • Recopila otros datos: alias, nombre, email (mínimos necesarios para identificación y compartición). • Envía una petición de alta al nodo, incluyendo: <ul style="list-style-type: none"> ◦ <i>user_id</i> ◦ clave pública ◦ alias (obligatorio) ◦ nombre (opcional) ◦ email (opcional) ◦ ... 4. El nodo: <ul style="list-style-type: none"> • Realiza una validación local: <ul style="list-style-type: none"> ◦ Verifica que no existe ya un usuario con ese <i>user_id</i> o alias. ◦ Verifica que todos los campos obligatorios están presentes y bien formados. ◦ Que la clave pública es válida (tamaño y formato esperado). ◦ Que el <i>user_id</i> corresponde al SHA-256 de la clave pública. • Si las validaciones son correctas: 	

CU01	Alta de usuario
	<ul style="list-style-type: none"> ○ Registra al nuevo usuario en su base de datos local. ○ Crea un evento user_registered firmado (por el nodo) que incluye: <ul style="list-style-type: none"> ■ user_id ■ clave pública ■ alias ■ fecha/hora de alta ■ ... ● Publica el evento en IOTA y obtiene el Block ID. ● El nodo publica un mensaje MQTT: <ul style="list-style-type: none"> ○ Tipo: user_registered ○ block_ID. <p>5. El resto de nodos que reciben el evento:</p> <ul style="list-style-type: none"> ● Descargan el contenido completo desde IOTA usando el Block ID. ● Verifican de forma similar al nodo local. ● Si todo es correcto registran también al usuario en su base de datos local.
Precondiciones	<ul style="list-style-type: none"> ■ El nodo debe estar conectado a IOTA y MQTT. ■ El navegador del usuario debe permitir almacenamiento seguro (para la clave privada). ■ El cliente debe ser capaz de derivar y almacenar localmente la clave privada protegida.
Postcondiciones	<ul style="list-style-type: none"> ■ El usuario queda registrado localmente en el nodo inicial. ■ El usuario queda registrado en el resto de nodos activos de la red. ■ El alias y la clave pública quedan disponibles para operaciones de compartición de ficheros.
Flujos alternativos	<p>FA01: El usuario ya existe</p> <ul style="list-style-type: none"> ■ Condición: <ul style="list-style-type: none"> ● Al validar localmente, se detecta que el user_id o alias ya existe. ■ Tratamiento: <ul style="list-style-type: none"> ● Rechazar el alta y registrar el error. ● Devolver error claro al usuario ("Alias o ID ya existente. Elija otro alias o recupere acceso si ya está registrado."). <p>FA02: Error al derivar claves en cliente web</p> <ul style="list-style-type: none"> ■ Condición: <ul style="list-style-type: none"> ● Falla la generación o derivación de claves en el navegador. ■ Tratamiento: <ul style="list-style-type: none"> ● Mostrar error y ofrecer reintento. <p>FA03: Error al publicar el evento en IOTA</p> <ul style="list-style-type: none"> ■ Condición: <ul style="list-style-type: none"> ● Fallo al enviar el evento de registro a IOTA. ■ Tratamiento: <ul style="list-style-type: none"> ● Registrar localmente el fallo. ● Reintentar N veces (con backoff exponencial). ● Si tras varios intentos sigue fallando, dejar el usuario registrado solo localmente con estado "pendiente de sincronización".

CU01	Alta de usuario
	<p>FA04: Error en la publicación de MQTT</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El evento se publica en IOTA, pero falla la notificación MQTT. ▪ Tratamiento: <ul style="list-style-type: none"> • Igual que en FA03: registrar error y reintentar, o marcar "pendiente". <p>FA05: Otro nodo rechaza el evento</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El evento recibido no pasa validaciones: • Firma inválida o evento corrupto. • user_id incorrecto. • Alias no válido o duplicado. ▪ Tratamiento: <ul style="list-style-type: none"> • Ignorar el registro de ese usuario. • (Opcional) Registrar incidente para auditoría de seguridad.

Fuente: elaboración propia.

CU02: Autenticación de usuario

La **Tabla 27** presenta el caso de uso extendido correspondiente al proceso de autenticación de usuario, detallando los pasos, actores y condiciones asociadas a su ejecución.

Tabla 27. CU02 - Autenticación de usuario

CU02	Autenticación de usuario
Actor principal	Usuario (cliente web)
Resumen	El usuario accede a la interfaz web de un nodo. El sistema valida el certificado local almacenado en el navegador. Si es válido y reconocido, el usuario queda autenticado. Si no, se ofrece la opción de alta.
Flujo principal	<ol style="list-style-type: none"> 1. El usuario accede a la página web de un nodo. 2. El cliente web: <ul style="list-style-type: none"> • Busca la clave privada almacenada en local (protegida por <i>passphrase</i>). • Sigue al usuario la <i>passphrase</i> para desbloquear la clave privada. • Usa la clave para firmar un desafío de autenticación generado por el servidor (por ejemplo: un <i>nonce</i> aleatorio). 3. El cliente envía al nodo: <ul style="list-style-type: none"> • El user_id (calculado como SHA-256 de la clave pública). • La firma del desafío. 4. El nodo: <ul style="list-style-type: none"> • Recupera la clave pública asociada al user_id en su base de datos. • Verifica la firma recibida con la clave pública.

CU02	Autenticación de usuario
	<ul style="list-style-type: none"> • Si la verificación es correcta, autentica al usuario. • Si la verificación falla, rechaza el acceso. <p>5. Una vez autenticado:</p> <ul style="list-style-type: none"> • Se genera una sesión segura o token de sesión. • El usuario puede empezar a interactuar normalmente con el sistema.
Precondiciones	<ul style="list-style-type: none"> ▪ El usuario debe haber completado el alta anteriormente. ▪ El navegador debe tener almacenadas su clave privada y su <i>passphrase</i>. ▪ El nodo debe tener la clave pública correspondiente al <i>user_id</i> registrado.
Postcondiciones	<ul style="list-style-type: none"> ▪ El usuario tiene una sesión activa en el nodo. ▪ El usuario puede realizar operaciones como subida de ficheros, compartición, descarga, etc. ▪ Se puede registrar (opcionalmente) un evento de acceso para trazabilidad (<i>file_accessed</i>).
Flujos alternativos	<p>FA01: No existe clave privada en el navegador</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El cliente no encuentra claves almacenadas localmente. ▪ Tratamiento: <ul style="list-style-type: none"> • Ofrecer al usuario la opción de registrarse (redirigir a Alta de Usuario). <p>FA02: Passphrase incorrecta</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El usuario introduce una <i>passphrase</i> incorrecta y no se puede desbloquear la clave privada. ▪ Tratamiento: <ul style="list-style-type: none"> • Mostrar error y ofrecer reintento. • (Opcional) Bloquear temporalmente tras varios intentos fallidos para seguridad. <p>FA03: No existe user_id registrado en el nodo</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El nodo no tiene el <i>user_id</i> en su base de datos. ▪ Tratamiento: <ul style="list-style-type: none"> • Rechazar el acceso. • Ofrecer al usuario la posibilidad de registrarse de nuevo. <p>FA04: Firma inválida</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • La firma enviada no es válida respecto al desafío emitido. ▪ Tratamiento: <ul style="list-style-type: none"> • Rechazar el acceso. • Mostrar error general de "Autenticación fallida".

Fuente: elaboración propia.

CU03: Alta de nodo

La **Tabla 28** presenta el caso de uso extendido correspondiente al alta de nodo, detallando los pasos, actores y condiciones asociadas a su ejecución.

Tabla 28. CU03 - Alta de nodo

CU03		Alta de nodo
Actor principal	Actor principal	
Resumen	<p>Cuando un nodo se inicia por primera vez, genera su clave privada y pública derivadas de una <i>passphrase</i> local. A continuación, se registra localmente y se da de alta en la red publicando un evento firmado en IOTA. El resto de los nodos recibirán el evento a través de MQTT, lo validarán y registrarán al nuevo nodo si la información es válida.</p>	
Flujo principal	<ol style="list-style-type: none">1. El nodo se inicia y comprueba si dispone de su clave privada:<ul style="list-style-type: none">• Si no existe:<ul style="list-style-type: none">◦ Solicita al administrador local una <i>passphrase</i>.◦ Deriva la clave privada de firma (ej. usando un KDF tipo PBKDF2, <i>scrypt</i> o <i>Argon2</i>).◦ Deriva la clave pública de verificación.◦ Calcula su <i>node_id</i> como SHA-256 de la clave pública.2. El nodo registra su información básica en su base de datos local:<ul style="list-style-type: none">• <i>node_id</i>• clave pública• fecha/hora de alta• versión de software• plataforma• etiquetas opcionales (tags)3. El nodo crea un evento firmado con su información de alta y lo publica en IOTA:<ul style="list-style-type: none">• Tipo de evento: <i>node_registered</i>• Contenido firmado: <i>node_id</i>, clave pública, metadatos básicos.4. Una vez publicado el evento, el nodo obtiene el <i>block_id</i> de IOTA.5. El nodo publica un mensaje MQTT que contiene:<ul style="list-style-type: none">• Tipo: <i>node_registered</i>• <i>block_id</i>.6. El resto de nodos suscritos a MQTT:<ul style="list-style-type: none">• Reciben el aviso.• Usan el <i>block_id</i> para recuperar el contenido del evento completo desde IOTA.• Verifican:<ul style="list-style-type: none">◦ Que la firma del evento es válida usando la clave pública incluida.◦ Que el <i>node_id</i> es el SHA-256 correcto de la clave pública.◦ Que ese <i>node_id</i> no existe ya en su base de datos local.◦ Que la versión de software cumple requisitos mínimos siquieres forzar compatibilidad.	

CU03	Alta de nodo
	<ul style="list-style-type: none"> • Si todo es correcto: <ul style="list-style-type: none"> ◦ Registran el nuevo nodo en su base de datos interna. ◦ Marcan el nodo como "activo". <p>Validaciones adicionales sugeridas</p> <ul style="list-style-type: none"> ▪ Verificar que la fecha del evento no sea superior a un umbral razonable (por ejemplo, no más de 1 día de diferencia respecto al reloj del sistema, para evitar registros "viejos" maliciosos). ▪ Rechazar registros donde el node_id ya exista. ▪ (Opcional) Comprobar que el tamaño y el formato de la clave pública es el esperado. ▪ (Opcional) Mantener un pequeño log o auditoría de nuevos registros de nodos.
Precondiciones	<ul style="list-style-type: none"> ▪ El nodo debe tener conexión a IOTA y a MQTT. ▪ El nodo debe tener acceso a almacenamiento local para su clave privada y su base de datos.
Postcondiciones	<ul style="list-style-type: none"> ▪ El nodo queda registrado en su propia base de datos local. ▪ El nodo queda registrado en el resto de los nodos de la red. ▪ La clave pública del nodo queda disponible para verificar mensajes futuros.
Flujos alternativos	<p>FA01: No se puede derivar la clave privada</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Error al generar o derivar la clave privada a partir de la <i>passphrase</i>. ▪ Tratamiento: <ul style="list-style-type: none"> • Mostrar error al usuario administrador local. • Abortarlo todo: el nodo no podrá continuar la inicialización. • Solicitar nueva <i>passphrase</i> o reiniciar el proceso. <p>FA02: Error al publicar el evento en IOTA</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El nodo no puede enviar el evento a la Tangle (falla la conexión o la publicación). ▪ Tratamiento: <ul style="list-style-type: none"> • Registrar el error localmente (log de error). • Intentar reintentar la publicación hasta un número limitado de veces (por ejemplo, 3 intentos con <i>backoff</i> exponencial). • Si sigue fallando: <ul style="list-style-type: none"> ◦ Marcar el nodo como "pendiente de registro". ◦ Programar un nuevo intento periódico en <i>background</i>. ◦ No publicar en MQTT todavía. <p>FA03: Error al publicar en MQTT</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El evento se publica correctamente en IOTA, pero falla el envío del Block ID al canal MQTT. ▪ Tratamiento: <ul style="list-style-type: none"> • Registrar error localmente. • Reintentar la publicación en MQTT hasta N intentos.

CU03	Alta de nodo
	<ul style="list-style-type: none"> • Si sigue fallando: <ul style="list-style-type: none"> ◦ Dejar una tarea periódica activa para seguir intentando mientras el nodo esté en ejecución. ◦ Marcar el nodo como "<i>registrado en IOTA, pendiente de anunciar en red</i>". <p>FA04: Otro nodo rechaza el evento (vista desde el lado de los nodos que reciben la alta)</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Al recibir un evento node_registered vía MQTT y descargar el contenido de IOTA: <ul style="list-style-type: none"> ◦ La firma es inválida. ◦ El node_id no corresponde al hash SHA-256 de la clave pública. ◦ El node_id ya existe en la base de datos. ◦ El evento contiene campos corruptos o incompletos. ▪ Tratamiento: <ul style="list-style-type: none"> • Registrar el error localmente para auditoría. • Ignorar el registro del nodo. • (Opcional) Publicar un evento node_registration_failed si se desea hacerlo más robusto (no implementado en la versión básica). <p>FA05: Error interno en la base de datos local</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Falla la grabación del nodo en la base de datos (problema de disco, corrupción, etc.). ▪ Tratamiento: <ul style="list-style-type: none"> • Registrar el error. • Intentar reintentar la operación. • Si persiste, marcar el nodo como "<i>registro incompleto</i>" y enviar alerta local o log.

Fuente: elaboración propia.

CU04: Actualización de estado de nodo

La **Tabla 29** presenta el caso de uso extendido correspondiente al proceso de actualización del estado de nodo, detallando los pasos, actores y condiciones asociadas a su ejecución.

Tabla 29. CU04 - Actualización de estado de nodo

CU04	Actualización de estado de nodo
Actor principal	Nodo (proceso automático y periódico)
Resumen	Cada nodo, de forma periódica, envía su información de estado actual (<i>uptime</i> , IP, puerto, versión de software, espacio disponible, etc.) a la red. El

CU04	Actualización de estado de nodo
	nodo publica un evento firmado en IOTA y notifica a través de MQTT a los demás nodos para que actualicen su registro de estado del nodo.
Flujo principal	<ol style="list-style-type: none"> 1. Cada nodo, a intervalos regulares (por ejemplo, cada 5 minutos): <ul style="list-style-type: none"> • Calcula y recopila su estado actual: <ul style="list-style-type: none"> ◦ uptime (tiempo en segundos desde el último arranque) ◦ ip_address (IP pública o visible para otros nodos) ◦ port (puerto TCP/UDP de servicio o comunicación) ◦ software_version (versión actual del nodo) ◦ (Opcional) free_space (espacio libre disponible en disco) ◦ (Opcional) load_average o estadísticas de uso del sistema • El nodo crea un evento node_status firmado: <ul style="list-style-type: none"> ◦ Incluye su node_id, timestamp actual, y la información anterior. • El nodo publica este evento firmado en IOTA. • Una vez obtenido el Block ID: <ul style="list-style-type: none"> ◦ Publica un mensaje corto en MQTT con el Block ID y tipo de evento node_status. <ol style="list-style-type: none"> 2. Los demás nodos: <ul style="list-style-type: none"> • Escuchan el mensaje MQTT. • Usan el Block ID para recuperar el contenido del evento completo desde IOTA. • Verifican: <ul style="list-style-type: none"> ◦ Que la firma del evento es válida utilizando la clave pública previamente registrada del nodo. ◦ Que el node_id corresponde a uno conocido y registrado. ◦ Validar que el timestamp no sea demasiado viejo o futuro (prevención de ataques de repetición). ◦ Verificar que la IP tiene un formato válido. 3. Si las validaciones son correctas: <ul style="list-style-type: none"> • Actualizan la información dinámica del nodo en su base de datos local: <ul style="list-style-type: none"> ◦ Última IP conocida ◦ Último puerto conocido ◦ Uptime ◦ Versión de software ◦ Timestamp de última actualización ◦ ...
Precondiciones	<ul style="list-style-type: none"> ▪ El nodo debe estar registrado previamente en la red. ▪ Debe tener acceso a IOTA y al bróker MQTT. ▪ Los nodos deben tener en su base local la clave pública del nodo emisor.
Postcondiciones	<ul style="list-style-type: none"> ▪ El estado actual del nodo queda actualizado en la base de datos local de todos los nodos activos. ▪ Permite tener siempre disponible información como: <ul style="list-style-type: none"> • ¿Está activo? • ¿Cuál es su IP y puerto de conexión? • ¿Cuánto uptime lleva? • ¿Qué versión de software está usando?
Flujos alternativos	FA01: Error al publicar en IOTA

CU04	Actualización de estado de nodo
	<ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El evento node_status no puede ser publicado en IOTA. ▪ Tratamiento: <ul style="list-style-type: none"> • Reintentar con <i>backoff</i> exponencial un número de veces limitado. • Si persiste el fallo, registrar localmente el error y seguir operando con el último estado conocido. <p>FA02: Error en la publicación de MQTT</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Se publica el evento en IOTA, pero falla la notificación MQTT. ▪ Tratamiento: <ul style="list-style-type: none"> • Registrar el error. • Reintentar la publicación MQTT varias veces. • Si persiste, no afectará demasiado, ya que otros nodos simplemente no se actualizarán hasta el siguiente latido. <p>FA03: Otro nodo rechaza el evento</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El evento recibido por otros nodos: <ul style="list-style-type: none"> ◦ Firma inválida. ◦ node_id desconocido. ▪ Tratamiento: <ul style="list-style-type: none"> • Ignorar la actualización para ese nodo. • Registrar localmente para auditoría si quieras detectar intentos maliciosos.

Fuente: elaboración propia.

CU05: Subida de fichero

La **Tabla 30** presenta el caso de uso extendido correspondiente al proceso de subida de fichero, detallando los pasos, actores y condiciones asociadas a su ejecución.

Tabla 30. CU05 - Subida de fichero

CU05	Subida de fichero
Actor principal	Usuario (cliente web)
Resumen	El usuario autenticado puede subir un nuevo fichero a través del cliente web. El fichero será cifrado previamente en el cliente, y la información del fichero (contenido cifrado, metadatos, clave compartida cifrada) será enviada al nodo. El nodo almacenará el fichero localmente, generará el identificador único (file_id) y publicará un evento para la red. Posteriormente otros nodos podrán replicarlo según las políticas de redundancia.

CU05	Subida de fichero
Flujo principal	<ol style="list-style-type: none"> 1. El usuario autenticado selecciona un fichero a subir mediante el cliente web. 2. El cliente web realiza las siguientes acciones: <ul style="list-style-type: none"> • Cifra el contenido del fichero con una clave compartida aleatoria (por ejemplo, AES-256). • Calcula el <code>file_id</code> como SHA-256 del contenido cifrado. • Cifra la clave compartida usando la clave pública del propio usuario. • Recoge metadatos: <ul style="list-style-type: none"> ◦ <code>Filename</code> ◦ Tamaño ◦ MIME type ◦ Hash SHA-256 previo al cifrado (opcional para verificación) ◦ Fecha de creación ◦ Otros campos como etiquetas (tags) si se desea 3. El cliente envía al nodo una petición de subida que contiene: <ul style="list-style-type: none"> • <code>file_id</code> • Contenido cifrado • Metadatos del fichero • Clave compartida cifrada para el propietario 4. El nodo: <ul style="list-style-type: none"> • Verifica que no exista ya un fichero con ese <code>file_id</code>. • Verifica que el <code>mimetype</code> (<code>text/*</code>, <code>image/*</code>, <code>application/pdf</code>, etc.), tamaño (ej. 100MB), etc. sean válidos • Guarda el contenido cifrado en su almacenamiento bajo <code>.storage/<file_id></code>. • Guarda los metadatos en <code>.meta/<file_id>.json</code>. • Crea una entrada en su sistema de ficheros virtual con el <code>filename</code>. 5. El nodo crea un evento <code>file_created</code> firmado que incluye: <ul style="list-style-type: none"> • <code>file_id</code> • <code>user_id</code> propietario • <code>filename</code> • <code>tamaño</code> • <code>mimetype</code> • fecha de creación • <code>tags</code> • nodos donde está replicado (inicialmente solo el propio). • ... 6. El nodo publica el evento en IOTA. 7. Tras obtener el Block ID: <ul style="list-style-type: none"> • Publica en MQTT un aviso de nuevo fichero creado, incluyendo el Block ID. 8. Otros nodos que reciban el evento: <ul style="list-style-type: none"> • Descargan los metadatos. • Realizan las mismas verificaciones. • Guarda los metadatos en <code>.meta/<file_id>.json</code>. • Crea una entrada en su sistema de ficheros virtual con el <code>filename</code>. • Evaluarán posteriormente si deben replicar el fichero según las políticas de redundancia.
Precondiciones	<ul style="list-style-type: none"> ▪ El usuario debe estar autenticado.

CU05	Subida de fichero
	<ul style="list-style-type: none"> ▪ El cliente web debe tener acceso a la clave pública del usuario. ▪ El nodo debe tener espacio de almacenamiento disponible.
Postcondiciones	<ul style="list-style-type: none"> ▪ El fichero cifrado queda almacenado localmente en el nodo de subida. ▪ El fichero es conocido en toda la red mediante evento distribuido. ▪ Otros nodos pueden iniciar procesos de replicación si corresponde.
Flujos alternativos	<p>FA01: El fichero ya existe</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El file_id calculado ya existe en el nodo. ▪ Tratamiento: <ul style="list-style-type: none"> • El sistema puede: <ul style="list-style-type: none"> ◦ bien rechazar la subida (ya existe). ◦ bien permitir crear una nueva entrada de visibilidad sin duplicar el contenido. <p>FA02: Error durante el cifrado o hash en el cliente</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Fallo al cifrar el fichero o calcular hashes. ▪ Tratamiento: <ul style="list-style-type: none"> • Mostrar error en el cliente. • Permitir reintentar. <p>FA03: Error al guardar el fichero en el nodo</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Fallo al escribir el fichero o los metadatos en disco. ▪ Tratamiento: <ul style="list-style-type: none"> • Rechazar la operación. • Devolver error al usuario. <p>FA04: Error al publicar en IOTA</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • No se puede registrar el evento file_created en IOTA. ▪ Tratamiento: <ul style="list-style-type: none"> • Reintentar publicación varias veces. • Si sigue fallando, marcar el fichero como "subido localmente pero pendiente de sincronizar". <p>FA05: Error en la publicación de MQTT</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Error en la notificación a otros nodos. ▪ Tratamiento: <ul style="list-style-type: none"> • Reintentar. • No afecta a la persistencia local, pero sí a la visibilidad de la red hasta que se solucione.

Fuente: elaboración propia.

CU06: Descarga de fichero

La **Tabla 31** presenta el caso de uso extendido correspondiente al proceso de descarga de fichero, detallando los pasos, actores y condiciones asociadas a su ejecución.

Tabla 31. CU06 - Descarga de fichero

CU06		Descarga de fichero
Actor principal	Usuario (cliente web)	
Resumen	Un usuario autenticado puede descargar un fichero previamente subido o compartido. El cliente web solicitará el fichero cifrado al nodo, lo descargará, recuperará la clave compartida cifrada, la descifrará con su clave privada local, y luego descifrará el contenido para obtener el fichero original.	
Flujo principal	<ol style="list-style-type: none"> 1. El usuario autenticado: <ul style="list-style-type: none"> • Navega su sistema de ficheros virtual (/users/{user_id}/). • Selecciona un fichero (por nombre visible, no por file_id). 2. El cliente web: <ul style="list-style-type: none"> • Sigue la descarga del fichero indicando su nombre (/users/{user_id}/nombre_de_fichero.pdf). 3. El nodo: <ul style="list-style-type: none"> • Resuelve la ruta a su file_id correspondiente. • Verifica que el usuario tiene permiso de acceso a ese fichero. • Recupera: <ul style="list-style-type: none"> ◦ El contenido cifrado del fichero desde .storage/<file_id>. ◦ El metadato asociado desde .meta/<file_id>.json, incluyendo la clave compartida cifrada para ese usuario. • Envía al cliente web: <ul style="list-style-type: none"> ◦ El contenido cifrado del fichero. ◦ La clave compartida cifrada para ese usuario. ◦ Opcionalmente, metadatos como tamaño y MIME type. ◦ Opcionalmente, impone restricciones de velocidad. 4. El cliente web: <ul style="list-style-type: none"> • Verifica que el contenido coincide con el ID • Descifra la clave compartida usando la clave privada local del usuario. • Descifra el contenido cifrado usando la clave compartida. • Verifica su integridad (checksum) y reconstruye el fichero original. 5. El usuario autenticado <ul style="list-style-type: none"> • Descarga el fichero descifrado. • O bien, lo abre en su navegador (según configuración). 	
Precondiciones	<ul style="list-style-type: none"> ▪ El usuario debe estar autenticado correctamente. ▪ El fichero debe existir y ser visible en el espacio virtual del usuario. ▪ El nodo debe tener acceso al contenido cifrado (.storage/) y metadatos (.meta/). 	
Postcondiciones	<ul style="list-style-type: none"> ▪ El usuario obtiene una copia local descifrada del fichero. 	

CU06	Descarga de fichero
Flujos alternativos	<ul style="list-style-type: none"> ▪ Se puede registrar (opcionalmente) un evento de acceso para trazabilidad (<code>file_accessed</code>). <p>FA01: El fichero no existe</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • La ruta solicitada no existe en el sistema de ficheros virtual. ▪ Tratamiento: <ul style="list-style-type: none"> • Devolver error 404 o mensaje de "<i>fichero no encontrado</i>". <p>FA02: El usuario no tiene acceso</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El usuario no figura en los permisos de acceso al fichero. ▪ Tratamiento: <ul style="list-style-type: none"> • Devolver error 403 o mensaje de "<i>acceso no autorizado</i>". <p>FA03: Error de descifrado de la clave compartida</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El cliente no puede descifrar la clave compartida (por ejemplo, <i>passphrase</i> incorrecta o claves corruptas). ▪ Tratamiento: <ul style="list-style-type: none"> • Mostrar error en cliente: "<i>Error al descifrar el fichero. Compruebe su clave privada.</i>" <p>FA04: El contenido cifrado o metadatos están corruptos</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El fichero o sus metadatos no pueden leerse correctamente. ▪ Tratamiento: <ul style="list-style-type: none"> • Devolver error claro al usuario: "<i>El fichero no se puede recuperar en este momento.</i>"

Fuente: elaboración propia.

CU07: Compartición de fichero

La **Tabla 32** presenta el caso de uso extendido correspondiente al proceso de compartición de fichero, detallando los pasos, actores y condiciones asociadas a su ejecución.

Tabla 32. CU07 - Compartición de fichero

CU07	Compartición de fichero
Actor principal	Usuario (cliente web)
Resumen	Un usuario autenticado puede compartir uno de sus ficheros con otro usuario de la red. La compartición implica autorizar el acceso cifrando la clave compartida para el destinatario y crear una nueva entrada visible en el sistema de ficheros virtual del usuario receptor, permitiéndole acceder al fichero.

CU07	Compartición de fichero
Flujo principal	<ol style="list-style-type: none"> 1. El usuario autenticado navega su espacio virtual (/users/{user_id}/) y selecciona el fichero que quiere compartir. 2. El cliente web: <ul style="list-style-type: none"> • Solicita al nodo una lista de usuarios disponibles (GET /users) para seleccionar destinatario. • El usuario selecciona el destinatario mediante su alias o nombre. • Selecciona la clave pública del usuario destinatario. • Recupera del nodo la clave compartida cifrada para el propietario. • Descifra localmente la clave compartida usando su clave privada. • Cifra de nuevo la clave compartida usando la clave pública del destinatario. • Envía una petición de compartición al nodo: <ul style="list-style-type: none"> ◦ Identificador del fichero (file_id). ◦ Identificador del usuario destino (target_user_id). ◦ Clave compartida cifrada para el destinatario. ◦ Nombre del fichero (nombre visible que se mostrará en su espacio virtual). 3. El nodo: <ul style="list-style-type: none"> • Registra la nueva autorización: añade el destinatario como usuario autorizado en los metadatos. • Crea una nueva entrada virtual (/users/{target_user_id}/{nombre_fichero}) apuntando al mismo file_id. • Crea un evento file_shared firmado que incluye: <ul style="list-style-type: none"> ◦ file_id ◦ owner ◦ target_user_id ◦ filename ◦ fecha de compartición ◦ claves simétricas cifradas • Publica el evento en IOTA y obtiene el Block ID. • Publica en MQTT un aviso de file_shared para sincronización de la red. 4. Los demás nodos: <ul style="list-style-type: none"> • Reciben el evento. • Verifican su validez (firma, usuarios conocidos, etc.). • Crean la misma entrada virtual en el espacio del destinatario en sus sistemas locales.
Precondiciones	<ul style="list-style-type: none"> ▪ El usuario debe estar autenticado. ▪ El fichero debe existir en el espacio del usuario. ▪ El usuario destinatario debe estar registrado en el sistema.
Postcondiciones	<ul style="list-style-type: none"> ▪ El fichero aparece visible en el espacio virtual del usuario destinatario. ▪ El destinatario puede descargar el fichero, descifrándolo, usando la clave compartida proporcionada.
Flujos alternativos	<p>FA01: El fichero no existe o no pertenece al usuario</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El fichero a compartir no existe o no es propiedad del usuario. ▪ Tratamiento:

CU07	Compartición de fichero
	<ul style="list-style-type: none"> • Rechazar la solicitud. • Devolver error claro. <p>FA02: Usuario destinatario no encontrado</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El usuario destino no existe en el sistema. ▪ Tratamiento: <ul style="list-style-type: none"> • Mostrar error en cliente: "<i>Usuario no encontrado</i>". <p>FA03: Error al cifrar o descifrar la clave compartida</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Falla el proceso de descifrado o cifrado de la clave compartida. ▪ Tratamiento: <ul style="list-style-type: none"> • Mostrar error en el cliente: "<i>Error de cifrado. No se puede completar la compartición</i>". <p>FA04: Error al registrar la entrada en el nodo</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Falla la creación de la entrada en el sistema de ficheros virtual. ▪ Tratamiento: <ul style="list-style-type: none"> • Registrar error en el nodo. • Informar al cliente del fallo. <p>FA05: Error en publicación de evento en IOTA o MQTT</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • No se puede publicar el evento <code>file_shared</code>. ▪ Tratamiento: <ul style="list-style-type: none"> • Reintentar. • Si persiste, marcar como "<i>compartición local pendiente de sincronización</i>".

Fuente: elaboración propia.

CU08: Listado de ficheros

La **Tabla 33** presenta el caso de uso extendido correspondiente al proceso de listado de ficheros, detallando los pasos, actores y condiciones asociadas a su ejecución.

Tabla 33. CU08 - Listado de ficheros

CU08	Listado de ficheros
Actor principal	Usuario (cliente web)
Resumen	El usuario autenticado puede solicitar al nodo la lista de los ficheros visibles en su espacio virtual. Esta lista incluye tanto los ficheros propios como los

CU08	Listado de ficheros
	ficheros compartidos por otros usuarios. El nodo responde navegando su sistema de entradas (entries) asociadas al user_id del usuario.
Flujo principal	<ol style="list-style-type: none"> 1. El usuario autenticado accede a la interfaz web del nodo. 2. El cliente web: <ul style="list-style-type: none"> • Sigue el listado de ficheros bajo su espacio /users/{user_id}/. • (Opcionalmente) Un subdirectorio si implementas carpetas en versiones futuras. 3. El nodo: <ul style="list-style-type: none"> • Identifica el user_id del usuario autenticado. • Navega el sistema de entradas (entries) correspondientes a ese user_id. • Recupera las entradas visibles, que contienen: <ul style="list-style-type: none"> ◦ Nombre del fichero o carpeta. ◦ Identificador interno (file_id) si es un fichero. ◦ Tipo (file o directory). ◦ Tamaño (si es un fichero). ◦ Fecha de creación. ◦ Propietario (puede ser él mismo o el usuario que compartió el fichero). ◦ Etiquetas (tags), si existen. • Responde con la lista de entradas disponibles. • (Opcionalmente) Limitar el tamaño de los listados (paginación si hay muchos ficheros). 4. El cliente web: <ul style="list-style-type: none"> • Muestra la lista en formato navegable al usuario: <ul style="list-style-type: none"> ◦ Nombre ◦ Tamaño ◦ Tipo ◦ Cualquier metadato útil. ◦ Opcionalmente, distinguir visualmente entre ficheros propios y compartidos. • (Opcionalmente) Ordenar o filtrar por varios criterios.
Precondiciones	<ul style="list-style-type: none"> ▪ El usuario debe estar autenticado. ▪ Debe existir al menos una entrada en su espacio virtual o bien el espacio estará vacío.
Postcondiciones	<ul style="list-style-type: none"> ▪ El usuario visualiza su espacio de ficheros actualizado.
Flujos alternativos	<p>FA01: No hay ficheros</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • No existen entradas en el espacio del usuario. ▪ Tratamiento: <ul style="list-style-type: none"> • Devolver una lista vacía. • Mostrar mensaje amigable: "No tienes ficheros aún." <p>FA02: Error en la recuperación de entradas</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Fallo al acceder al sistema de ficheros virtual. ▪ Tratamiento: <ul style="list-style-type: none"> • Manejar el error y devolver una respuesta adecuada.

CU08	Listado de ficheros
	<ul style="list-style-type: none"> • Devolver error general al cliente: "No se pudo recuperar la lista de ficheros." • Registrar error internamente. <p>FA03: Error de autenticación</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El usuario no está autenticado o la sesión caducó. ▪ Tratamiento: <ul style="list-style-type: none"> • Redirigir al usuario al proceso de <i>login</i>.

Fuente: elaboración propia.

CU09: Renombrado de fichero virtual

La **Tabla 34** presenta el caso de uso extendido correspondiente al proceso de renombrado de fichero virtual, detallando los pasos, actores y condiciones asociadas a su ejecución.

Tabla 34. CU09 - Renombrado de entrada

CU09	Renombrado de entrada
Actor principal	Usuario (cliente web)
Resumen	El usuario autenticado puede renombrar uno de sus ficheros virtuales visibles en su espacio. El renombrado afecta únicamente a la entrada (entry) en el sistema de ficheros virtual, no al contenido real del fichero (file_id) ni a los datos cifrados. Renombrar es simplemente mover una entrada simbólica de un nombre a otro.
Flujo principal	<ol style="list-style-type: none"> 1. El usuario autenticado visualiza su lista de ficheros. 2. El cliente web ofrece opción de renombrar un fichero (por ejemplo: botón derecho > "Renombrar"). 3. El usuario introduce el nuevo nombre deseado para el fichero. 4. El cliente web envía una petición de renombrado al nodo que incluye <ul style="list-style-type: none"> • Ruta actual (/users/{user_id}/nombre_viejo.ext). • Nuevo nombre deseado (nuevo_nombre.ext). 5. El nodo: <ul style="list-style-type: none"> • Verifica que la entrada original existe y pertenece al usuario. • Verifica que no existe ya una entrada con el nuevo nombre en el mismo espacio. • Actualiza el nombre en el sistema de entradas (mueve o renombra). • Registra el evento file_renamed en IOTA y obtiene el Block ID. Debe contener: <ul style="list-style-type: none"> ◦ file_id ◦ user_id ◦ filename ◦ new_filename ◦ timestamp

CU09	Renombrado de entrada
	<ul style="list-style-type: none"> Notificar por MQTT para informar al resto de nodos. <p>6. El cliente actualiza la vista para reflejar el nuevo nombre.</p> <p>7. Los demás nodos:</p> <ul style="list-style-type: none"> Reciben el aviso y recuperan el evento completo de IOTA. Verifican la firma y consistencia. Aplican el renombrado en su sistema local de entradas para el usuario afectado.
Precondiciones	<ul style="list-style-type: none"> El usuario debe estar autenticado. El fichero a renombrar debe existir y pertenecer al usuario.
Postcondiciones	<ul style="list-style-type: none"> El fichero aparece en el sistema de ficheros virtual con el nuevo nombre. El contenido cifrado (<code>file_id</code>) no cambia.
Flujos alternativos	<p>FA01: El fichero no existe</p> <ul style="list-style-type: none"> Condición: <ul style="list-style-type: none"> El fichero no existe. Tratamiento: <ul style="list-style-type: none"> Devolver error claro: "<i>Fichero no encontrado.</i>" <p>FA02: Ya existe un fichero con el nuevo nombre</p> <ul style="list-style-type: none"> Condición: <ul style="list-style-type: none"> Ya existe un fichero con el nuevo nombre. Tratamiento: <ul style="list-style-type: none"> Devolver error claro: "<i>Ya existe un fichero con ese nombre en esta ubicación</i>". <p>FA03: Error interno al renombrar</p> <ul style="list-style-type: none"> Condición: <ul style="list-style-type: none"> Se ha producido cualquier otro tipo de error al gestionar el sistema de ficheros. Tratamiento: <ul style="list-style-type: none"> Devolver error general y registrar log de fallo.

Fuente: elaboración propia.

CU10: Borrado de fichero

La **Tabla 35** presenta el caso de uso extendido correspondiente al proceso de borrado de fichero, detallando los pasos, actores y las condiciones asociadas a su ejecución.

Tabla 35. CU10 - Borrado de fichero

CU10	Borrado de entrada
Actor principal	Usuario (cliente web)
Resumen	El usuario autenticado puede borrar una entrada virtual asociada a un fichero. Si el fichero ya no tiene ninguna entrada apuntándolo (ningún

CU10	Borrado de entrada
	usuario lo referencia), el contenido real (datos cifrados) y metadatos pueden ser eliminados físicamente.
Flujo principal	<ol style="list-style-type: none"> 1. El usuario autenticado visualiza su lista de ficheros. 2. El cliente web ofrece opción de borrar un fichero (por ejemplo: botón derecho > "Eliminar"). 3. El usuario confirma la eliminación. 4. El cliente web envía una petición de borrado al nodo que incluye: <ul style="list-style-type: none"> • Ruta del fichero a eliminar (/users/{user_id}/nombre.ext). 5. El nodo: <ul style="list-style-type: none"> • Verifica que el fichero existe y pertenece al usuario. • Elimina la entrada del sistema de ficheros virtual. • Comprueba si el file_id correspondiente ya no tiene ninguna entrada visible en ningún espacio de usuario local. • Si es así, puede: <ul style="list-style-type: none"> ◦ Borrar el contenido cifrado .storage/<file_id>. ◦ Borrar los metadatos .meta/<file_id>.json. • Generar un evento file_deleted para sincronización en toda la red que incluye: <ul style="list-style-type: none"> ◦ file_id ◦ user_id ◦ Nombre del fichero borrado ◦ Timestamp • Publica el evento en IOTA y obtiene Block ID. • Publica un aviso en MQTT para sincronizar la eliminación. 6. El cliente actualiza la vista para reflejar el nuevo nombre. 7. Los demás nodos: <ul style="list-style-type: none"> • Reciben el aviso y recuperan el evento completo de IOTA. • Verifican la firma y consistencia. • Eliminan la entrada en su sistema local de entradas para el usuario afectado. • Si corresponde, también eliminan el fichero si ya no tiene más referencias locales.
Precondiciones	<ul style="list-style-type: none"> ▪ El usuario debe estar autenticado. ▪ El fichero debe existir y ser visible en su espacio.
Postcondiciones	<ul style="list-style-type: none"> ▪ La entrada del fichero desaparece del sistema de ficheros virtual del usuario. ▪ El contenido real puede ser eliminado si ya no está referenciado.
Flujos alternativos	<p>FA01: El fichero no existe</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El fichero no existe. ▪ Tratamiento: <ul style="list-style-type: none"> • Devolver error claro: "Fichero no encontrado". <p>FA02: Error interno al eliminar la entrada</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El fichero no existe. ▪ Tratamiento: <ul style="list-style-type: none"> • Devolver error general y registrar fallo.

Fuente: elaboración propia.

CU11: Replicación de fichero

La **Tabla 36** presenta el caso de uso extendido correspondiente al proceso de replicación de fichero, detallando los pasos, actores participantes y condiciones asociadas a su ejecución.

Tabla 36. CU11 - Replicación de fichero

CU11 Replicación de fichero	
Actor principal	Nodo (proceso automático)
Resumen	Cuando un nodo detecta un nuevo fichero creado en la red, evalúa si debe replicarlo localmente en función de las políticas de redundancia configuradas (por ejemplo, mantener siempre N copias en diferentes nodos). Si decide replicarlo, descarga el contenido cifrado desde otro nodo o desde IOTA, lo almacena localmente, y actualiza sus registros internos.
Flujo principal	<ol style="list-style-type: none">1. El nodo:<ul style="list-style-type: none">• Recibe un evento <code>file_created</code> a través de MQTT.• Descarga el contenido completo del evento desde IOTA.• Analiza el evento:<ul style="list-style-type: none">◦ Extrae información: <code>file_id</code>, tamaño, propietario, etc.◦ Comprueba cuántos nodos (replicas) ya han almacenado el fichero (<code>replica_nodes</code> en metadatos).• Evalúa su política de replicación (por ejemplo: "cada fichero debe estar replicado al menos en 3 nodos").• Si el número de réplicas actuales es insuficiente, replicar localmente:<ul style="list-style-type: none">◦ Sigue el fichero cifrado a uno de los nodos que lo tiene (por ejemplo, usando un API REST GET <code>/files/{file_id}</code>)◦ Lo guarda el fichero en <code>.storage/<file_id></code>.◦ Actualiza los metadatos en <code>.meta/<file_id>.json</code> y añade su propio <code>node_id</code> al campo <code>replica_nodes</code>.• Genera un evento <code>file_replicated</code> firmado que incluye:<ul style="list-style-type: none">◦ <code>file_id</code>.◦ <code>replica_node_id</code>.◦ <code>timestamp</code>.• Publica el evento en IOTA.• Notifica por MQTT. <ol style="list-style-type: none">2. El resto de nodos:<ul style="list-style-type: none">• Reciben el aviso.• Actualizan sus registros internos para saber que un nuevo nodo almacena ese <code>file_id</code>.
Precondiciones	<ul style="list-style-type: none">▪ El nodo debe estar registrado y activo.▪ El nodo debe tener espacio suficiente disponible para almacenar el fichero.▪ El fichero debe estar autorizado para replicación (no archivos privados bloqueados si en el futuro implementas esa lógica).

CU11	Replicación de fichero
Postcondiciones	<ul style="list-style-type: none"> ▪ El fichero cifrado queda almacenado también en el nodo replicador. ▪ El campo <code>replica_nodes</code> se actualiza en todos los nodos para reflejar la nueva réplica.
Flujos alternativos	<p>FA01: Error al descargar el fichero</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Fallo al solicitar el fichero a otro nodo. ▪ Tratamiento: <ul style="list-style-type: none"> • Reintentar con otros nodos. • Si persiste, registrar error y programar reintentos más tarde. <p>FA02: Espacio insuficiente</p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • El nodo no tiene suficiente espacio disponible. ▪ Tratamiento: <ul style="list-style-type: none"> • No realizar replicación. • Registrar aviso de espacio insuficiente. <p>FA03: Error al publicar evento <code>file_replicated</code></p> <ul style="list-style-type: none"> ▪ Condición: <ul style="list-style-type: none"> • Falla publicación en IOTA o MQTT. ▪ Tratamiento: <ul style="list-style-type: none"> • Reintentar. • Si no es posible, marcar el fichero como "replicado localmente, pendiente de anunciar".

Fuente: elaboración propia.

Anexo F. Modelo de clases

El modelo de clases representa las entidades principales del sistema y sus relaciones estáticas. Permite visualizar la estructura interna del software y sirve como base para la implementación de sus componentes.

Los modelos presentados a continuación fueron diseñados en una etapa temprana con el objetivo de definir la arquitectura general del sistema. Pueden no reflejar con exactitud la implementación final. Algunas estructuras se han ido adaptado y/o simplificado durante la fase de desarrollo hasta la solución final.

Código UML

```
@startuml

class Usuario {
    +String user_id
    +String clave_publica
    +String alias
    +String nombre
    +String email
    +List<String> tags
    +Integer version
}

class Nodo {
    +String node_id
    +String clave_publica
    +String ip
    +Integer puerto
    +Long uptime
    +String software_version
    +String platform
    +List<String> tags
    +Integer version
    +Timestamp ultimo_heartbeat
}

class Fichero {
    +String file_id
    +String owner
    +Long tamano
    +String mime_type
    +String hash
    +List<String> tags
    +List<String> replica_nodes
    +Timestamp fecha_creacion
    +Integer version
}
```

```
class Entrada {  
    +String path  
    +String file_id  
    +String user_id  
    +String tipo  
    +String nombre  
    +Timestamp fecha_creacion  
}  
  
class Evento {  
    +String event_id  
    +String tipo  
    +JSON contenido  
    +String firma  
    +String publicador_id  
    +Timestamp timestamp  
    +String block_id  
}  
  
Usuario "1" -- "*" Entrada : posee  
Fichero "1" -- "*" Entrada : referenciado por  
Nodo "1" -- "*" Evento : publica  
  
Evento "1" -- "1" Usuario : relacionado con  
Evento "1" -- "1" Nodo : relacionado con  
Evento "1" -- "1" Fichero : relacionado con  
  
@enduml
```

Clases

Atributos principales de cada clase.

Usuario

```
user_id: String (SHA-256 de la clave pública)  
clave_publica: String  
alias: String  
nombre: String (opcional)  
email: String (opcional)  
tags: List<String> (opcional)  
version: Integer
```

Nodo

```
node_id: String (SHA-256 de la clave pública)  
clave_publica: String  
ip: String  
puerto: Integer  
uptime: Long  
software_version: String  
platform: String
```

```
tags: List<String> (opcional)  
version: Integer  
ultimo_heartbeat: Timestamp
```

Fichero

```
file_id: String (SHA-256 del contenido cifrado)  
owner: String  
tamaño: Long  
mime_type: String  
hash: String (opcional)  
tags: List<String> (opcional)  
replica_nodes: List<String>  
fecha_creacion: Timestamp  
version: Integer
```

Entrada

```
path: String (ruta virtual, ej.: /users/{user_id}/documento.pdf)  
file_id: String  
user_id: String (usuario propietario de la entrada)  
tipo: Enum { file, directory }  
nombre: String  
fecha_creacion: Timestamp
```

Evento

```
event_id: String (ID único del evento)  
tipo: Enum { node_registered, node_status, user_registered, file_created,  
file_shared, file_deleted, file_replicated, file_renamed }  
contenido: JSON  
firma: String  
publicador_id: String (node_id o user_id según quien publica)  
timestamp: Timestamp  
block_id: String (referencia en IOTA)
```

Anexo G. Diagrama de actividades

El presente anexo incluye el diagrama de actividades. Se representa de forma gráfica el flujo secuencial de acciones que se ejecutan durante una operación típica con el objetivo de ayudar a comprender la lógica del comportamiento dinámico del sistema.

Los diagramas que se muestran a continuación se diseñaron en una etapa temprana para definir la arquitectura general del sistema y pueden no reflejar con exactitud la implementación final.

Subida de fichero

La **Figura 41** representa el flujo completo de subida y propagación de un fichero. Incluye desde su selección por parte del usuario, hasta la sincronización entre nodos. Describe cómo se cifra el contenido, calcula el identificador del archivo y publica el evento `file_created` en IOTA para que el resto de nodos repliquen la información.

```
@startuml

start

:Seleccionar fichero en cliente web;
:Generar clave simetrica y cifrar contenido;
:Calcular file_id (SHA-256 del contenido cifrado);
:Cifrar clave simetrica con clave publica del usuario;
:Enviar fichero cifrado, metadatos y nombre al nodo;

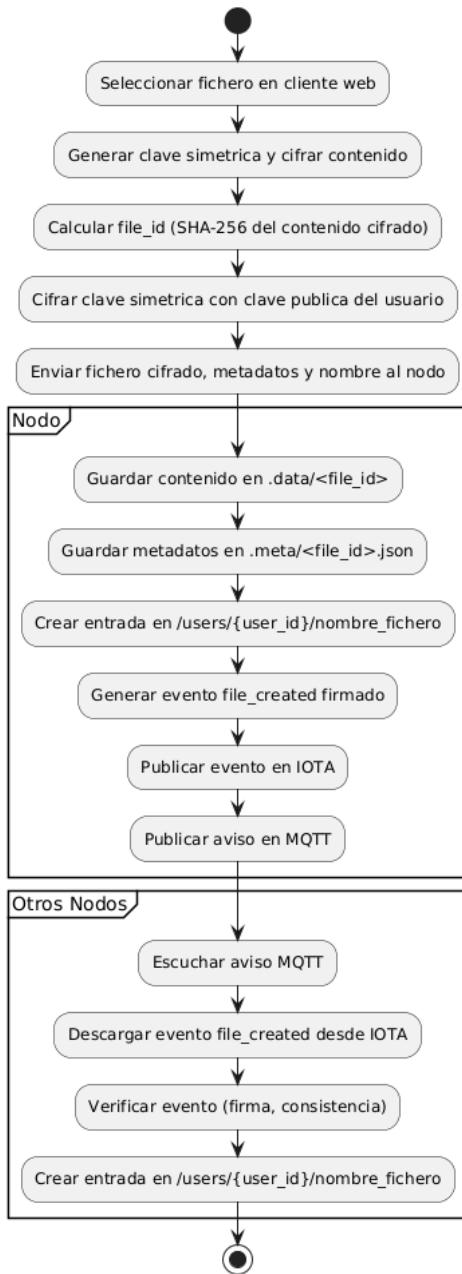
partition Nodo {
    :Guardar contenido en .storage/<file_id>;
    :Guardar metadatos en .meta/<file_id>.json;
    :Crear entrada en /users/{user_id}/nombre_fichero;
    :Generar evento file_created firmado;
    :Publicar evento en IOTA;
    :Publicar aviso en MQTT;
}

partition Otros Nodos {
    :Escuchar aviso MQTT;
    :Descargar evento file_created desde IOTA;
    :Verificar evento (firma, consistencia);
    :Crear entrada en /users/{user_id}/nombre_fichero;
}

stop

@enduml
```

Figura 41. Subida de fichero



Fuente: elaboración propia.

Descarga de fichero

La **Figura 42** detalla el flujo de descarga de ficheros. Incluye desde la solicitud del usuario hasta la recuperación y descifrado del contenido. El nodo verifica los permisos, recupera los datos cifrados y la clave compartida con la que el cliente descifra el contenido.

@startuml

```

start

:Seleccionar fichero en cliente web;
:Enviar solicitud de descarga al nodo (ruta del fichero);

partition Nodo {
    :Verificar permisos de acceso;
    :Resolver ruta a file_id;
    :Recuperar contenido cifrado (.storage/<file_id>);
    :Recuperar clave compartida cifrada para el usuario;
    :Enviar contenido cifrado y clave cifrada al cliente;
}

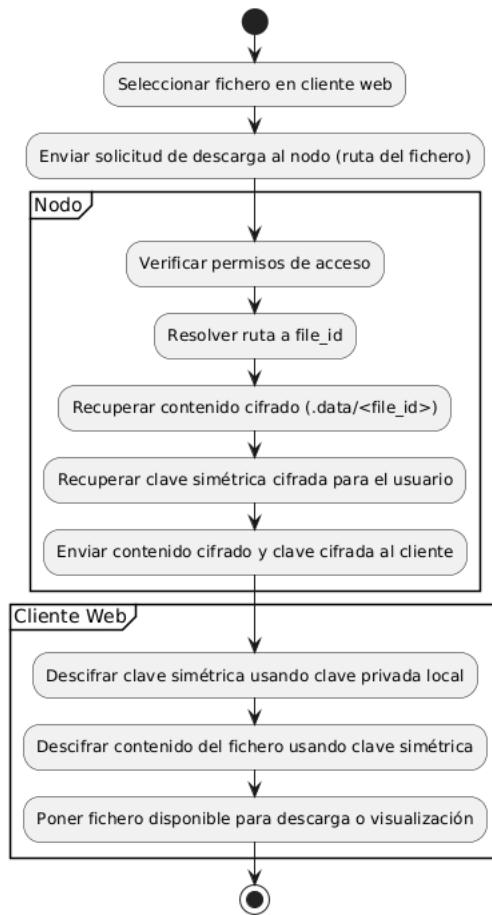
partition Cliente Web {
    :Descifrar clave compartida usando clave privada local;
    :Descifrar contenido del fichero usando clave compartida;
    :Poner fichero disponible para descarga o visualización;
}

stop

@enduml

```

Figura 42. Descarga de fichero



Fuente: elaboración propia.

Compartición de fichero

La **Figura 43** representa el flujo completo de compartición de ficheros. El usuario selecciona un destinatario y cifra la clave compartida del fichero específicamente para él, enviando una solicitud de compartición al nodo. Tras registrar la autorización y crear la entrada virtual correspondiente, se publica el evento `file_shared` para que el resto de nodos lo repliquen de forma consistente.

```
@startuml

start

:Seleccionar fichero a compartir en cliente web;
:Solicitar lista de usuarios disponibles;
:Seleccionar destinatario;
:Solicitar clave pública del destinatario;
:Recuperar clave compartida cifrada del fichero;
:Descifrar clave compartida usando clave privada local;
:Cifrar clave compartida para destinatario;
:Enviar solicitud de compartición al nodo;

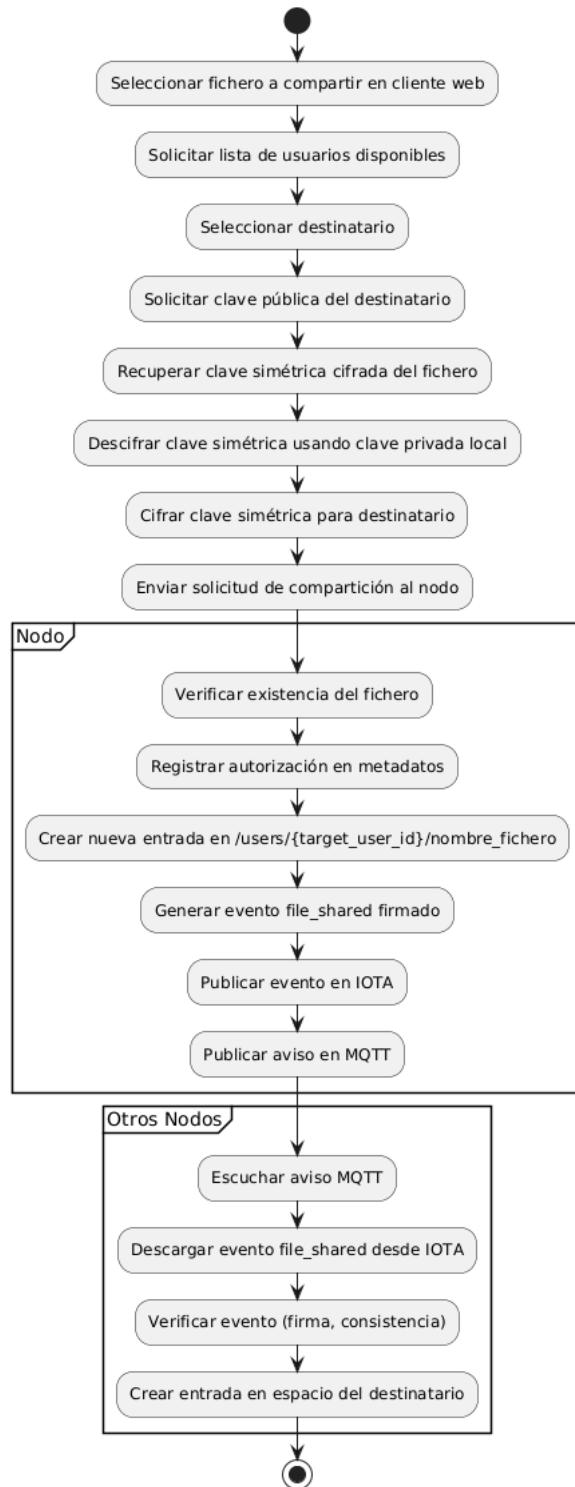
partition Nodo {
    :Verificar existencia del fichero;
    :Registrar autorización en metadatos;
    :Crear nueva entrada en /users/{target_user_id}/nombre_fichero;
    :Generar evento file_shared firmado;
    :Publicar evento en IOTA;
    :Publicar aviso en MQTT;
}

partition Otros Nodos {
    :Escuchar aviso MQTT;
    :Descargar evento file_shared desde IOTA;
    :Verificar evento (firma, consistencia);
    :Crear entrada en espacio del destinatario;
}

stop

@enduml
```

Figura 43. Compartición de fichero

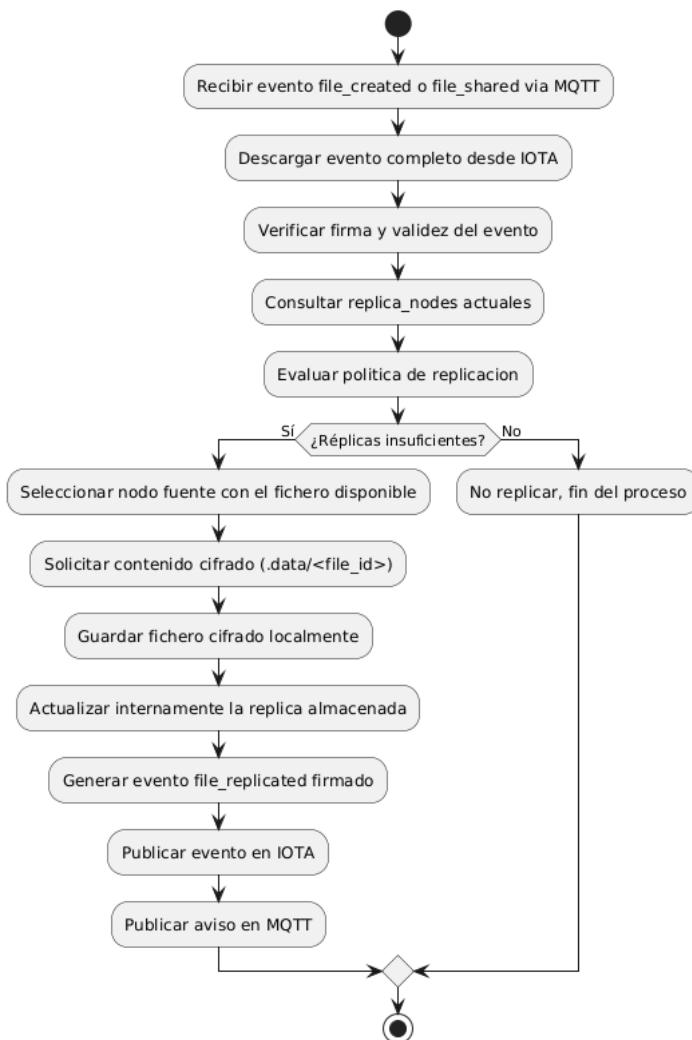


Fuente: elaboración propia.

Replicación de fichero

La **Figura 44** describe el flujo de replicación de archivos. Se activa tras la recepción de un evento `file_created` o `file_shared`. El nodo analiza si hay réplicas suficientes, según la política definida; en caso negativo, solicita el contenido cifrado a otro nodo, lo almacena localmente y publica un evento `file_replicated` para que se actualice el estado global.

Figura 44. Replicación de fichero



Fuente: elaboración propia.

@startuml

start

:Recibir evento file_created o file_shared via MQTT;
:Descargar evento completo desde IOTA;
:Verificar firma y validez del evento;

```
:Consultar replica_nodes actuales;
:Evaluar politica de replicacion;

if (¿Rélicas insuficientes?) then (Sí)
    :Seleccionar nodo fuente con el fichero disponible;
    :Solicitar contenido cifrado (.storage/<file_id>);
    :Guardar fichero cifrado localmente;
    :Actualizar internamente la replica almacenada;
    :Generar evento file_replicated firmado;
    :Publicar evento en IOTA;
    :Publicar aviso en MQTT;
else (No)
    :No replicar, fin del proceso;
endif

stop

@enduml
```

Alta de usuario

La **Figura 45** describe el flujo completo de registro de un nuevo usuario. Desde el cliente web se genera un par de claves y se envía una solicitud al nodo. Éste valida la unicidad del user_id y publica el evento correspondiente para que el resto de nodos sincronicen la nueva identidad.

```
@startuml

start

:Acceder al cliente web;
:Intentar autenticacion;
:Detectar que usuario no esta registrado;
:Solicitar passphrase;
:Generar claves publica/privada y user_id;
:Solicitar alias y otros datosopcionales;
:Enviar solicitud de alta al nodo;

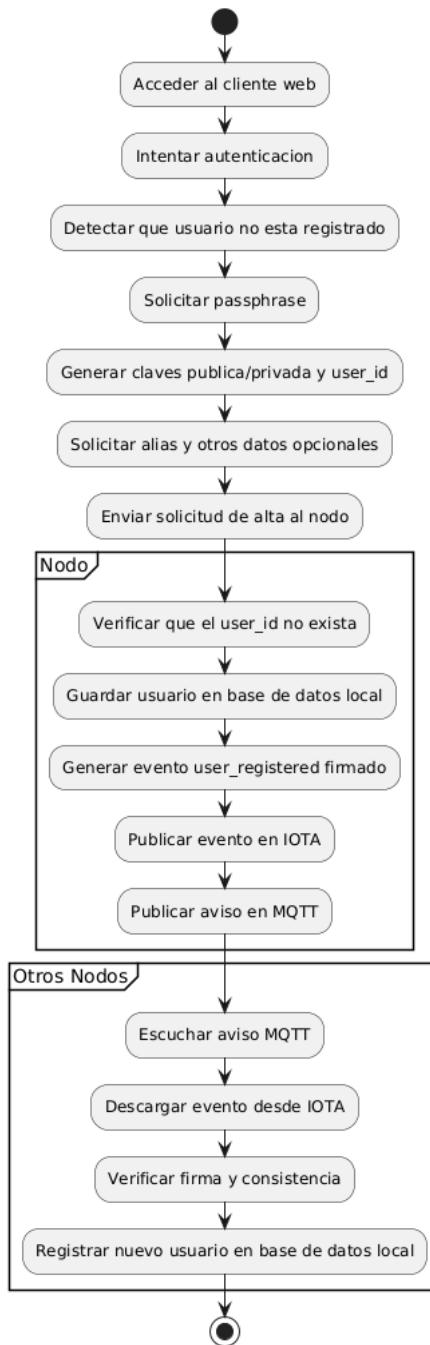
partition Nodo {
    :Verificar que el user_id no exista;
    :Guardar usuario en base de datos local;
    :Generar evento user_registered firmado;
    :Publicar evento en IOTA;
    :Publicar aviso en MQTT;
}

partition Otros Nodos {
    :Escuchar aviso MQTT;
    :Descargar evento desde IOTA;
    :Verificar firma y consistencia;
    :Registrar nuevo usuario en base de datos local;
}

stop
```

@endum1

Figura 45. Alta de usuario

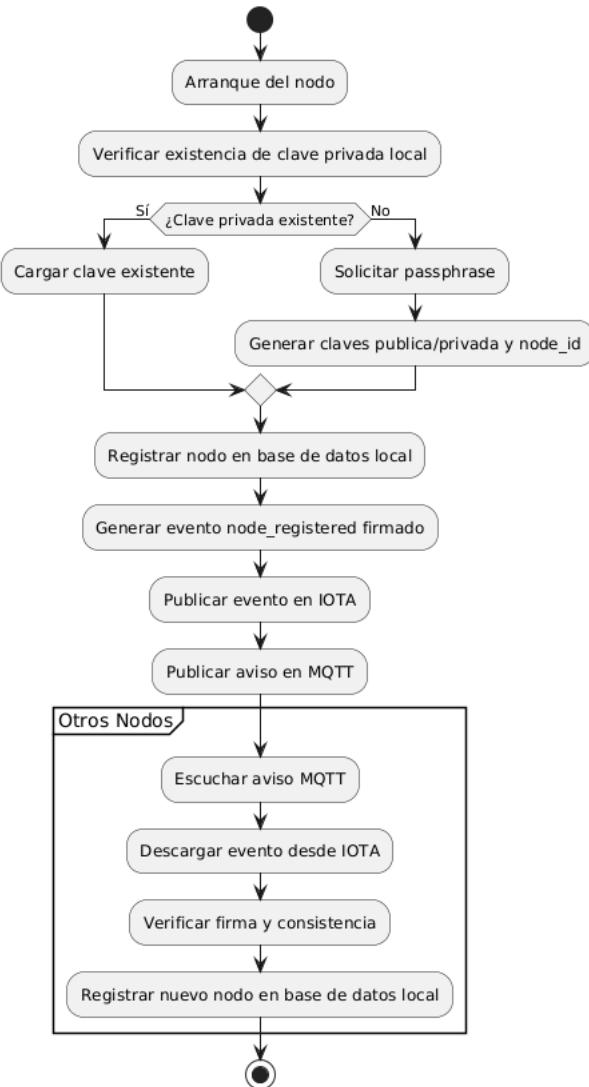


Fuente: elaboración propia.

Alta de nodo

La **Figura 46** ilustra el flujo de arranque de un nodo. Si no tiene asociada una identidad previa, se genera un nuevo par de claves y el `node_id`. A continuación, el nodo se registra localmente y publica un evento `node_registered` para que el resto de nodos lo integren en la red.

Figura 46. Alta de nodo



Fuente: elaboración propia.

```

@startuml
start
:Arranque del nodo;
:Verificar existencia de clave privada local;
if (¿Clave privada existente?) then (Sí)
  
```

```
:Cargar clave existente;
else (No)
:Solicitar passphrase;
:Generar claves publica/privada y node_id;
endif

:Registrar nodo en base de datos local;
:Generar evento node_registered firmado;
:Publicar evento en IOTA;
:Publicar aviso en MQTT;

partition Otros Nodos {
:Escuchar aviso MQTT;
:Descargar evento desde IOTA;
:Verificar firma y consistencia;
:Registrar nuevo nodo en base de datos local;
}

stop

@enduml
```

Anexo H. Modelo de base de datos

Este anexo incluye UML de relaciones entre tablas, definición de tablas y código SQL para SQLite.

Relaciones UML

Se incluye la descripción de los diagramas de relaciones UML que representan las asociaciones entre las principales entidades del sistema.

```
@startuml

title dfs3 - Data Model (UML)

class User {
    +user_id: String
    +public_key: String
    +alias: String
    +name: String
    +email: String
    +tags: String
    +version: int
}

class Node {
    +node_id: String
    +alias: String
    +hostname: String
    +public_key: String
    +ip: String
    +port: int
    +uptime: int
    +software_version: String
    +platform: String
    +tags: String
    +version: int
    +total_space: int
    +last_seen: datetime
    +creation_date: datetime
}

class File {
    +file_id: String
    +owner: String
    +size: int
    +mime_type: String
    +hash: String
    +tags: String
    +creation_date: datetime
    +version: int
}

class Entry {
```

```

+path: String
+file_id: String
+owner: String
+creation_date: datetime
}

class Event {
    +event_id: String
    +type: String
    +timestamp: datetime
    +data: String
    +signature: String
    +issuer_node_id: String
}

' Relaciones
User "1" -- "0..*" File : owns
User "1" -- "0..*" Entry : has
Node "1" -- "0..*" Event : emits
File "1" -- "0..*" Entry : referenced by
User "1" -- "0..*" Node : associated with
Event "0..*" --> Node : issued by

@enduml

```

Tablas

Descripción de las tablas que dan soporte a la aplicación con sus correspondientes campos.

Tabla 37. Tabla 'users'

Campo	Tipo	Descripción
user_id	VARCHAR(64)	Identificador único del usuario (SHA-256 de su clave pública).
public_key	TEXT	Clave pública del usuario en formato PEM/base64.
alias	VARCHAR(32)	Alias público del usuario, único y legible.
name	VARCHAR(100)	Nombre completo del usuario (opcional).
email	VARCHAR(100)	Dirección de correo electrónico del usuario (opcional).
tags	TEXT	Lista opcional de etiquetas clasificadorias (en formato JSON).
version	INTEGER	Número de versión para control de cambios y sincronización.
creation_date	TIMESTAMP	Fecha y hora de registro del usuario.
last_seen	TIMESTAMP	Fecha y hora de última conexión.

Fuente: elaboración propia.

Tabla 38. Tabla 'nodes'

Campo	Tipo	Descripción
-------	------	-------------

node_id (*)	VARCHAR(64)	Identificador único del nodo (SHA-256 de su clave pública).
alias	VARCHAR(50)	Para facilitar la identificación del nodo
hostname	VARCHAR(100)	Hostname del nodo (devuelto por el S.O.)
public_key	TEXT	Clave pública del nodo.
ip	VARCHAR(45)	Dirección IP del nodo (compatible con IPv4 e IPv6).
port	INTEGER	Puerto en el que el nodo expone su API REST.
uptime	INTEGER	Tiempo de actividad del nodo (en segundos).
software_version	VARCHAR(20)	Versión del software ejecutándose en el nodo.
platform	VARCHAR(50)	Sistema operativo o plataforma (por ejemplo, Linux/armv7).
tags	TEXT	Lista de etiquetas descriptivas del nodo (en formato JSON).
version	INTEGER	Número de versión para seguimiento de actualizaciones.
total_space	INTEGER	Espacio libre (en bytes)
creation_date	TIMESTAMP	Fecha y hora de registro del nodo
last_seen	TIMESTAMP	Fecha y hora del último mensaje recibido desde este nodo.

Fuente: elaboración propia.

Tabla 39. Tabla 'files'

Campo	Tipo	Descripción
file_id (*)	VARCHAR(64)	Identificador único del fichero (hash SHA-256 del contenido cifrado).
owner	VARCHAR(64)	Identificador del usuario propietario del fichero.
size	INTEGER	Tamaño del fichero cifrado (en bytes).
mime_type	VARCHAR(100)	Tipo MIME del fichero (por ejemplo, application/pdf).
hash	VARCHAR(32)	Hash SHA-256 del contenido original (sin cifrar), opcional.
tags	TEXT	Lista de etiquetas descriptivas (en formato JSON).
creation_date	TIMESTAMP	Fecha de creación del fichero.
version	INTEGER	Número de versión para gestión de metadatos.

Fuente: elaboración propia.

Tabla 40. Tabla 'entries'

Campo	Tipo	Descripción
path	VARCHAR(255)	Ruta virtual o nombre del fichero visible para el usuario.
file_id	VARCHAR(64)	Referencia al fichero cifrado real.
owner	VARCHAR(64)	Usuario propietario de esta entrada virtual.

creation_date	TIMESTAMP	Fecha de creación de la entrada.
----------------------	-----------	----------------------------------

Fuente: elaboración propia.

Tabla 41. Tabla 'events'

Campo	Tipo	Descripción
block_id (*)	VARCHAR(64)	Identificador único del evento, block ID en IOTA.
event_type	VARCHAR(32)	Tipo de evento (por ejemplo, file_created, node_status, etc.).
timestamp	TIMESTAMP	Momento en el que ocurrió el evento.
node_id	VARCHAR(64)	Identificador del nodo que emitió el evento.

Fuente: elaboración propia.

Código SQL (SQLite)

Se incluye el código SQL utilizado para la creación de las tablas de la base de datos en SQLite. Estas tablas sirven de soporte persistente al sistema y permiten almacenar la información estructurada relativa a usuarios, nodos y eventos.

```
-- Table: users
CREATE TABLE users (
    user_id TEXT PRIMARY KEY,
    public_key TEXT NOT NULL,
    alias TEXT NOT NULL,
    name TEXT,
    email TEXT,
    tags TEXT,
    creation_date TIMESTAMP NOT NULL,
    last_seen TIMESTAMP NOT NULL,
    version INTEGER NOT NULL
);

-- Table: nodes
CREATE TABLE nodes (
    node_id TEXT PRIMARY KEY,
    alias TEXT,
    hostname TEXT,
    version INTEGER NOT NULL,
    public_key TEXT NOT NULL,
    platform TEXT,
    software_version TEXT,
    uptime INTEGER NOT NULL,
    total_space INTEGER DEFAULT 0,
    ip TEXT NOT NULL,
    port INTEGER NOT NULL,
    tags TEXT,
    creation_date TIMESTAMP NOT NULL,
    last_seen TIMESTAMP NOT NULL
);
```

```
);

-- Table: events
CREATE TABLE events (
    block_id TEXT PRIMARY KEY,
    event_type TEXT NOT NULL CHECK (event_type IN (
        'user_registered',
        'user_joined_node',
        'node_registered',
        'node_status',
        'file_created',
        'file_deleted',
        'file_shared',
        'file_copied',
        'file_replicated',
        'file_renamed',
        'file_accessed'
    )),
    timestamp TIMESTAMP NOT NULL,
    node_id TEXT NOT NULL,
    FOREIGN KEY (node_id) REFERENCES nodes(node_id)
);

CREATE INDEX idx_events_event_type ON events(event_type);
CREATE INDEX idx_events_node_id ON events(node_id);
```

Anexo I. Lógica de replicación de ficheros

Se describe la lógica implementada para la replicación de ficheros entre nodos, determinando cuándo, cómo y en qué condiciones un fichero cifrado es replicado en otros nodos de la red. El objetivo es mejorar la disponibilidad, reducir la latencia en el acceso y proporcionar tolerancia a fallos.

Algoritmo

A continuación, pasamos a describir el algoritmo de replicación que implementan los nodos al detectar un nuevo fichero:

1. Cada nodo construye localmente una lista de nodos activos, filtrando por la combinación de varios criterios (parametrizables):
 - *Uptime* superior a **60 minutos**.
 - Última señal (`node_status`) recibida hace menos de **10 minutos**.
 - Disponer de espacio suficiente para almacenar el fichero.
2. Ordena esos nodos:
 - Primero por espacio libre disponible (mayor primero).
 - Si hay empate en espacio, por `node_id` en orden alfabético ascendente (criterio de desempate determinista).
3. Selecciona los **3 primeros nodos** resultantes.
4. Si el nodo propio está en esa lista:
 - Sigue una copia del fichero y la almacena en local.
 - Publica un evento `file_replicated` en la red.

Consideraciones

- No pasa nada si hay más copias de las deseadas inicialmente (por cache, accesos, etc.).
No se implementará en esta primera versión una política automática de purga o reducción de copias.

- El evento `file_replicated` servirá para localización, trazabilidad y para que otros nodos puedan ver qué ficheros tienen ya buenas garantías de réplica.
- En caso de que no haya espacio suficiente, se generará un mensaje de error.
- El algoritmo no tiene en cuenta aspectos avanzados como la latencia (cercanía entre nodos) y tiende a favorecer los nodos con más espacio (algo que podría conducir a cierta centralización).

Criterios y parámetros de replicación

Cada nodo decide de forma local y autónoma si debe o no replicar un fichero nuevo, evaluando los siguientes criterios:

Filtrado de nodos candidatos

Se construye una lista de nodos activos a partir de los siguientes requisitos, definidos como parámetros configurables del sistema:

- Uptime superior a 60 minutos: este umbral excluye nodos efímeros o inestables, y favorece la replicación en dispositivos con mayor permanencia operativa. Idealmente, este parámetro debería ser lo más alto posible (del orden de días e incluso semanas). No obstante, al tratarse de un entorno de I+D se ha asignado inicialmente un valor más bajo.
- Recepción reciente de un evento node_status (menos de 10 minutos): este criterio asegura que los nodos considerados están activos en el momento de la evaluación.
- Disponibilidad de espacio suficiente: se verifica que el nodo tiene capacidad para almacenar el fichero, evitando fallos posteriores en el proceso de replicación.

Ordenación determinista de candidatos

Los nodos que cumplen los criterios anteriores se ordenan siguiendo dos reglas:

- Espacio libre disponible (orden descendente): se prioriza la utilización de nodos con mayor capacidad, promoviendo un reparto equilibrado de la carga de almacenamiento.

- Identificador de nodo alfabético (`node_id` ascendente): este segundo criterio funciona como desempate determinista y garantiza que todos los nodos lleguen a la misma decisión sin necesidad de comunicación adicional.

Selección de nodos replicadores.

Se seleccionan los **tres primeros nodos** de la lista ordenada para almacenar una réplica del fichero. Este valor fijo responde a una combinación de criterios prácticos y teóricos:

- Redundancia tolerante a fallos: con tres réplicas distribuidas en nodos distintos, el sistema es capaz de resistir la caída simultánea de hasta dos nodos sin pérdida de datos, cubriendo una amplia gama de escenarios de desconexión temporal, reinicio o fallo permanente, habituales en entornos IoT.
- Simplicidad y eficiencia: aumentar el número de réplicas incrementa la tolerancia a fallos, pero también el uso de ancho de banda, almacenamiento y sincronización. Tres copias representan un equilibrio razonable entre robustez y coste, especialmente en redes con recursos limitados.
- Compatibilidad con modelos de quorum: la elección de un número impar de réplicas favorece la implementación futura de esquemas de consenso o validación parcial, como quorum mayoritario (2 de 3), sin necesidad de reconfiguración.
- Inspiración en sistemas distribuidos consolidados: plataformas como *Hadoop Distributed File System* (HDFS) y *Ceph* emplean tres réplicas por bloque de datos. Esta práctica, validada a gran escala, refuerza su adecuación como valor por defecto.
- Resiliencia en topologías pequeñas: en redes donde el número total de nodos puede ser bajo (por ejemplo, 5-10), mantener al menos tres copias independientes asegura cobertura geográfica y probabilidad alta de recuperación, incluso con nodos desconectados temporalmente.

Cabe destacar que la elección de tres réplicas responde a un valor empírico y configurable. Debería revisarse en función del tamaño de la red, la disponibilidad media de los nodos, el tipo de ficheros almacenados y la tolerancia al fallo deseada. En escenarios con alta

disponibilidad y muchos nodos, podría reducirse; en redes críticas o inestables, aumentarse. Se ha diseñado para que sea fácilmente ajustable sin comprometer la coherencia general.

Lógica de auto inclusión

Si el nodo evaluador se encuentra entre los tres seleccionados, se auto incluye como replicador, procede a solicitar el fichero al nodo fuente y lo almacena localmente. Tras completar la réplica, publica un evento `file_replicated` con el que notifica su disponibilidad al resto de nodos.

Anexo J. Límite de tamaño de fichero

El sistema establece un límite superior al tamaño de los ficheros permitidos con el objetivo de garantizar un funcionamiento eficiente y estable en entornos compuestos por nodos ligeros o de bajo consumo (como dispositivos IoT). Este umbral actúa como mecanismo preventivo frente a situaciones de saturación de memoria, almacenamiento o red.

La justificación técnica para este límite se basa en varios factores que pasamos a discutir:

- **Restricciones del hardware:** muchos nodos participantes (por ejemplo, dispositivos IoT o máquinas virtuales ligeras) disponen de recursos limitados, tanto en almacenamiento como en memoria RAM. Ficheros excesivamente grandes pueden comprometer su operatividad.
- **Coste computacional:** las operaciones de cifrado, fragmentación, replicación y reconstrucción de ficheros crecen en complejidad con el tamaño. Limitarlo permite mantener una carga razonable y predecible.
- **Replicación y comunicación en red:** en una red distribuida, cada fichero debe ser replicado entre varios nodos. Ficheros grandes ralentizan la propagación y aumentan el tráfico en redes que, en muchos casos, pueden tener limitaciones de ancho de banda.

Por tanto, se propone un límite por defecto de 10 MB para despliegues domésticos o educativos y hasta 100 MB en redes controladas con nodos más potentes. No obstante, este valor es configurable y puede ajustarse en función de las características del entorno, el número de nodos activos y la tolerancia al fallo deseada.

Anexo K. API REST

A continuación, se describen los *endpoints* implementados. Por su arquitectura modular, la API se puede ampliar fácilmente.

Autenticación y registro

El sistema utiliza un modelo de autenticación basado en criptografía asimétrica. Cada usuario se identifica mediante el hash SHA-256 de su clave pública en hexadecimal (`user_id`) y demuestra su identidad firmando un reto (`challenge`) proporcionado por el nodo. Este mecanismo garantiza la autenticación segura y la identidad sin necesidad de almacenar credenciales sensibles en los nodos distribuidos (no confiables por definición).

POST /auth/register

Registra un nuevo usuario.

- Petición (body):

```
{  
  "user_id": "e0c180e5d9bd6a7a4cad07b89e033890d14c96fa38bdfe847182376826da2d7f",  
  "alias": "test-user",  
  "name": "Test User",  
  "email": "test.user@dom.com",  
  "public_key": "TcRRzZeSUHMgIHs9Erw05uji202G1M/Tn+YqG19ktnk=",  
  "tags": [ "test", ... ]  
}
```

- Respuesta:

201 Created

```
{  
  "user_id": "e0c180e5d9..."  
}
```

POST /auth/challenge

Inicia sesión usando una prueba de posesión de clave privada (fase 1, petición de reto).

- Petición (body):

```
{
```

```
        "user_id": "e0c180e5d9..."  
    }
```

- El servidor responde con un reto (*challenge*) aleatorio que deberá firmarse:

```
{  
    "challenge": "<reto_base64>"  
}
```

POST /auth/verify

Segundo paso, verificación (fase 2). El cliente responde firmando el *challenge* con su clave privada.

- Petición (body):

```
{  
    "user_id": "abc123...",  
    "signature": "base64signature"  
}
```

- El servidor verifica la firma usando la clave pública asociada. Respuesta:

200 OK

```
{  
    "access_token": "abc123..."  
}
```

Gestión de usuarios

Permite gestionar los usuarios registrados a través de un conjunto de *endpoints* bajo /users. Se ha incluido solo la información mínima para que la aplicación funcione, incluyendo el user_id, alias y la clave pública del usuario (necesaria para cifrar claves simétricas en el proceso de compartición segura de archivos). Todas las operaciones requieren autenticación previa.

GET /users

Listar usuarios registrados.

- Respuesta:

[

```
[  
  {  
    "user_id": "ab3f2c...",  
    "alias": "nacho",  
    "public_key": "abc123..."  
  },  
  ...  
]
```

GET /users/{user_id}

Obtiene información pública de un usuario. Solo muestra la información básica necesaria (alias y public_key).

- Respuesta:

```
{  
  "user_id": "...",  
  "alias": "nacho",  
  "public_key": "abc123..."  
}
```

Nodos

Consultar información sobre los nodos registrados. De momento solo se ha incluido la información mínima necesaria: listado de nodos con información genérica sobre cada uno de ellos, incluyendo su identificador y alias.

El registro, actualización y monitorización del estado de los nodos (espacio libre, tiempo activo, direcciones IP, etc.) se realiza fuera de la API REST, mediante eventos publicados en la red IOTA / MQTT. La separación permite la sincronización entre nodos sin requerir de comunicación REST directa entre ellos.

GET /nodes

Lista los nodos conocidos por el sistema. Puede servir para visualización, auditoría o para redundancia / disponibilidad.

- Respuesta:

```
[  
  {  
    "node_id": "abc123...",  
    "alias": "node1",  
    "public_key": "abc123..."  
  },  
  ...  
]
```

]
...

GET /nodes/{node_id}

Obtiene información detallada de un nodo específico. Útil para consulta directa desde interfaces o para desarrollar herramientas de gestión y/o monitorización.

- Respuesta:

```
{  
    "node_id": "abc123...",  
    "alias": "node1",  
    "public_key": "abc123..."  
}
```

Ficheros

Gestionar tanto los ficheros reales como su representación virtual en el espacio de usuario. La subida de ficheros se realiza mediante POST /files. Cada fichero está identificado globalmente por su file_id. La mayoría de las operaciones requieren de autenticación previa.

GET /files

Lista las entradas en el directorio raíz del usuario. Equivalente a /<raiz>/users/{user_id}.

- Respuesta:

```
{  
    "entries": [  
        {  
            "name": "informe.pdf",  
            "file_id": "abc123...",  
            "size": 123456,  
            "mimetype": "application/pdf",  
            "creation_date": "2025-04-21T10:03:42Z"  
        },  
        ...  
    ]  
}
```

POST /files

Sube un nuevo fichero. Las rutas se procesan como entradas virtuales dentro de la carpeta de cada usuario (/<raiz>/users/{user_id}/). El sistema:

1. Calcula el SHA-256 del contenido cifrado y lo compara con el fichero subido.
 2. Si no coinciden: 400 Bad Request con error de integridad.
 3. Guarda contenido en .storage/<file_id>.dat.
 4. Crea metadatos en .meta/<file_id>.json.
 5. Registra la clave compartida (cifrada con la pública del propietario) para descifrado posterior.
 6. Registra un fichero virtual en la estructura de carpetas que apunta a .meta/<file_id>.json.
- Petición (`multipart/form-data`): la **Tabla 42** contiene la lista de campos enviados como metadatos.

Tabla 42. Campos de POST /files

Campo	Tipo	Descripción
data	binario	Contenido cifrado del fichero.
filename	string	Ruta completa donde debe aparecer el fichero (ej. tfg.pdf).
file_id	string	SHA-256 del contenido cifrado.
size	integer	Tamaño del fichero en bytes.
mimetype	string	Tipo MIME del fichero original.
sha256	string	SHA-256 del contenido antes de cifrar (integridad).
iv	string	Vector de inicialización usado en el cifrado del fichero.
tags	array (op)	Lista de etiquetas opcionales para el fichero.
authorized_users	array	Listado de usuarios con acceso al fichero. Para cada entrada está su identificador único (<code>user_id</code>), la clave compartida con la que se cifró el fichero (<code>encrypted_key</code>), cifrada a su vez con la clave pública del usuario y el vector de inicialización necesario para descifrar dicha clave (<code>iv</code>).

Fuente: elaboración propia.

Ejemplo:

```
{
  "filename": "README.md",
  "file_id": "f2bf762e73ff4410ec6d0b5b8d6aaeebaa0aa40dc5bd074c9a9dc85c7a4da3cd",
  "size": 3000,
  "mimetype": "text/markdown",
```

```
"sha256": "a68c2e53eb52e828ded921c22f2f1f5a6052bfa83f8dcb1f25c5a466c369186d",
"iv": "YjLF0fxDGYuIAK5G",
"tags": ["test", "dfs3"],
"authorized_users": [
  {
    "user_id": "e638b76feb3bb73f...",
    "encrypted_key": "+lGUKfg/wYqsW/2Cugs5LY/...",
    "iv": "WYUjVrJfeGGeXOha"
  }
]
```

- Respuesta (201 CREATED):

```
{
  "status": "stored"
}
```

GET /files/{filename}

Devuelve el contenido (cifrado) del fichero identificado por `filename` y registra un evento `file_accessed` de auditoría. Los metadatos necesarios para descifrar el fichero se adjuntan como cabeceras HTTP con formato X-DFS3-Xxxx:

- Respuesta (headers):

```
X-DFS3-File-ID: file_id (sha256 del fichero cifrado)
X-DFS3-Owner: user_id del propietario
X-DFS3-Size: tamaño del fichero
X-DFS3-IV: iv del fichero cifrado
X-DFS3-SHA256: sha256 del fichero sin cifrar
X-DFS3-Mimetype: mimetype
X-DFS3-Encrypted-Key: encrypted_key del usuario
X-DFS3-IV-Key: iv_key del usuario
X-DFS3-Public-Key: clave pública del propietario
```

GET /files/{file_id}/meta

Devuelve los metadatos asociados al fichero identificado por `file_id`.

- Respuesta:

```
{
  "file_id": "bc42c94eac7df123f1a1b4f890eefb2de91828be4f73fc1e0fa723b1b7e45c80",
  "owner": "3a142f...a9f",
  "size": 102348,
  "mimetype": "application/pdf",
  "creation_date": "2025-04-21T10:03:42Z",
  "tags": ["tfg", "pdf"],
  "replica_nodes": ["node01", "node07"]
}
```

GET /files/{file_id}/data

Devuelve el contenido cifrado del fichero identificado por `file_id`. Esta función no requiere de autenticación (los datos están cifrados y hay que conocer el hash). Se ha hecho así para simplificar la replicación entre nodos.

PATCH /files/{filename}

Renombra un fichero virtual (cambia el nombre bajo el que aparece).

- Petición (body):

```
{  
    "new_name": "tfg_final.pdf"  
}
```

- Respuesta:

```
{  
    "status": "renamed"  
}
```

DELETE /files/{filename}

Elimina la entrada virtual del usuario. Si no quedan referencias, se elimina también el fichero real con datos.

- Respuesta (mínima):

```
{  
    "status": "deleted"  
}
```

POST /files/share

Crea una entrada en el espacio virtual del usuario destino.

- Petición (body):

```
{  
    "filename": "informe_tfg.pdf",  
    "authorized_users": [  
        {  
            "user_id": "e638b76feb3bb73f...",  
            "encrypted_key": "+lGUKfg/wYqsW/2Cugs5LY/...",  
            "iv": "WYUjVrJfeGGeXOha"  
    ]  
}
```

```
        }
    ]
}
```

- Respuesta (mínima):

```
{
  "status": "shared"
}
```

Eventos

Permite consultar información sobre los eventos registrados. De momento solo información mínima necesaria.

GET /events

Lista los eventos registrados hasta la fecha. Se utiliza para auditoría y para sincronización del estado inicial en nodos nuevos.

- Respuesta:

```
[
  {
    "block_id": "0xabc123...",
    "timestamp": "2025-04-21T10:03:42Z",
    "event_type": "file_created",
    "node_id": "abc123..."
  },
  ...
]
```

GET /events/{node_id}

Obtiene información detallada de un evento específico. Actualmente no se utiliza, aunque se ha incluido para mantener la API lo más homogénea posible.

- Respuesta:

```
{
  "block_id": "0xabc123...",
  "timestamp": "2025-04-21T10:03:42Z",
  "event_type": "file_created",
  "node_id": "abc123..."
}
```

Anexo L. Metainformación de fichero

En la **Tabla 43** se detalla la estructura y contenido de la metainformación almacenada en formato JSON para cada fichero.

Tabla 43. Metainformación de fichero

Campo	Descripción
file_id	Identificador único del fichero, calculado como el hash SHA-256 del contenido cifrado.
size	Tamaño del fichero cifrado en bytes.
creation_date	Fecha y hora de creación del fichero, en formato ISO 8601.
sha256	Hash SHA-256 del fichero original sin cifrar, útil para verificación de integridad antes del cifrado.
iv	Vector de inicialización criptográfico usado para descifrar la información contenida en el fichero.
owner	Identificador del propietario, calculado como el hash SHA-256 de su clave pública.
authorized_users	Lista de usuarios con acceso al fichero, incluyendo para cada uno su identificador y la clave compartida cifrada.
mimetype	Tipo MIME del fichero original (por ejemplo, application/pdf, image/png, etc.).
tags	Lista de etiquetas opcionales para clasificar o facilitar la búsqueda de ficheros.
replica_nodes	Lista de nodos que almacenan una copia cifrada del fichero.
version	Número de versión de los metadatos, útil para control de cambios o sincronización.

Fuente: elaboración propia.

Ejemplo:

```
{
  "file_id": "09419c00e2c8e1f2e4906d23b205398339367273b02e269c5b95a56daeffe2a4",
  "owner": "fd8534506ad3b8a7262a3f82d28f28c70f72d1b4d74e401b28342d73b70a870d",
  "size": 19,
  "iv": "XVLMDi3nOnHoBgJ/",
  "sha256": "2a94a200762a30cc332703c4ef4b1c6af40e76e1a9214e3bfc6d2049878f6c9e",
  "mimetype": "text/plain",
  "tags": ["test", "dfs3"],
  "authorized_users": [
    {
      "user_id": "fd8534506ad3b8a7262a3f82d28f28c70f72d1b4d7...",
      "encrypted_key": "hK8RylHP2WJffrr9ueoDSQVrmLKbet6aXzfm8L08VpAzR...",
      "iv": "/kSpZhSNm4a027bR"
    }
  ]
}
```

```
        },
    ],
    "creation_date": "2025-05-11T07:27:43.964391+00:00",
    "version": 1,
    "replica_nodes": [
        "e8f985af468e17eed94d21343ef069403ef21b29f2e420b218fc5d565c8f1787"
    ]
}
```

Anexo M. Definición de eventos

Se describen los distintos tipos de eventos utilizados para coordinar la comunicación entre nodos y mantener la coherencia del sistema distribuido. Cada evento representa una acción que se publica en la red IOTA en formato JSON. Su definición permite a cada nodo interpretar, validar y aplicar cambios de forma autónoma y sincronizada. Todos los eventos siguen una estructura fija que se compone de los campos descritos en la **Tabla 44** y un payload específico por tipo:

Tabla 44. Estructura general de un evento

Campo	Tipo	Descripción
event_type	string	Tipo de evento: node_registered, file_created, etc.
timestamp	string (ISO 8601)	Fecha y hora UTC de creación del evento.
node_id	string (SHA-256)	Identificador del nodo que ha emitido el evento (hash de su clave pública).
protocol	string	Versión del protocolo utilizado (ej. dfs3/1.0).
payload	object	Contenido específico del evento. Varía según el event_type.
signature	string (base64)	Firma digital del nodo sobre el contenido del payload.

Fuente: elaboración propia.

Ejemplo de evento:

```
{  
  "event_type": "node_registered",  
  "timestamp": "2025-05-02T06:16:29.261997+00:00",  
  "node_id": "4e97faf4ec9fb5454a4b0a54b26ce6a4a682264e80f2b79ccab2d16e2870ac9f",  
  "protocol": "dfs3/1.0",  
  "payload": {  
    "alias": "mi-nodo",  
    "hostname": "node.dfs3.net",  
    "version": "dfs3-node/0.1.0",  
    "public_key": "R1u/z1W+Ee/F1Hpi+/GgXVY4KmIBl0mcAjjOMCuimnc=",  
    "platform": "Linux-6.8.0-45-generic-x86_64-with-glibc2.39",  
    "software_version": "dfs3-node/0.1.0",  
    ...  
  },  
  "signature": "FnOAhQUE..."  
}
```

Usuarios

Eventos relacionados con la actividad del usuario, incluyendo su registro y vinculación a un nodo. La **Tabla 45** contiene un listado de todos los eventos definidos para la gestión y tratamiento de usuarios.

Tabla 45. Eventos relativos a usuario

Tipo de evento	Descripción
<code>user_registered</code>	Registro de un nuevo usuario en la red. Incluye clave pública, alias, email, etc. Firmado por el nodo que lo registra.
<code>user_joined_node</code>	El usuario se ha autenticado desde un nuevo nodo. Útil para visibilidad o auditoría.

Fuente: elaboración propia.

Ejemplos de payload:

```
// user_registered
{
  "user_id": "abcdef01234567890",
  "alias": "nacho",
  "name": "Nacho Bravo",
  "email": "nacho@dom.com",
  "public_key": "...",
  "tags": ["test", "verified"],
  "version": 1
}

// user_joined_node
{
  "user_id": "7f9d...b1",
  "challenge": "...",      // Debería contener el timestamp
  "signature": "...",      // Firma del challenge
  "public_key": "...",    // Opcional, para simplificar validación
}
```

Nodos

Eventos asociados a los nodos del sistema. Permiten registrar su estado, actualización y participación en la red distribuida. La **Tabla 46** contiene un listado de los tipos de eventos definidos para la gestión de nodos.

Tabla 46. Eventos relativos a nodo

Evento	Descripción
node_registered	Alta inicial del nodo en la red. Publica su clave pública, alias, plataforma, capacidad total, etc.
node_status	Actualización periódica (<i>heartbeat</i>) con IP actual, espacio libre, <i>uptime</i> , puerto, estado operativo, etc.

Fuente: elaboración propia.

Ejemplos de payload:

```
// node_registered
{
  "alias": "mi-nodo",
  "hostname": "node.dfs3.net",
  "version": "dfs3-node/0.1.0",
  "public_key": "R1u/z1W+Ee/F1Hpi+/GgXVY4KmIBl0mcAjjOMCuimnc=",
  "platform": "Linux-6.8.0-45-generic-x86_64-with-glibc2.39",
  "software_version": "dfs3-node/0.1.0",
  "uptime": 17715119,
  "total_space": 269407227904,
  "ip": "194.124.43.131",
  "port": "1234",
  "tags": ["test", "dfs3"],
  "version": 1
}

// node_status
{
  "ip": "192.168.1.42",
  "port": 8080,
  "uptime": 123456,
  "total_space": 750000000000
}
```

Ficheros

Eventos vinculados a la gestión de ficheros, incluyendo su creación, compartición, eliminación y registro de acceso. La **Tabla 47** contiene un listado de los tipos de evento definidos para la gestión y tratamiento de ficheros.

Tabla 47. Eventos relativos a fichero

Evento	Descripción
file_created	El usuario sube un nuevo archivo. Se publica su hash, metadatos iniciales y la clave compartida cifrada para el propietario.

Evento	Descripción
file_shared	Se otorga acceso de lectura a otro usuario, cifrando la clave compartida de cifrado de datos con su clave pública.
file_accessed	Un usuario accede (descarga) un archivo. Útil para auditoría y estadísticas.
file_deleted	El usuario elimina la referencia local al archivo. El contenido solo se borra si no quedan más entradas virtuales.
file_renamed	Se modifica la entrada virtual (nombre) de un archivo sin cambiar su contenido.
file_replicated	Un nodo informa de que ha replicado el contenido del archivo localmente.

Fuente: elaboración propia.

Ejemplos de payload:

```
// file_created
{
  "user_id": "abcdef0123456789...",
  "file_id": "9c3f3c3b3050f82f6b557...",
  "filename": "informe.pdf",
  "size": 1048576,
  "mimetype": "application/pdf",
  "sha256": "d41d8cd98f00b204e9800998ecf8427e...",
  "iv": "abcdef01234567890...",
  "authorized_users": [
    {
      "user_id": "abcdef0123456789...",
      "encrypted_key": "abcdef0123456789...",
      "iv": "abcdef0123456789...",
    },
    ...
  ],
  "tags": ["proyecto", "confidencial"]
}

// file_shared
{
  "user_id": "abcdef0123456789...",
  "file_id": "a1b2c3...e8f9",
  "filename": "informe_tfg.pdf",
  "authorized_users": [
    {
      "user_id": "e638b76feb3bb73f...",
      "encrypted_key": "+lGUKfg/wYqsW/2Cugs5LY/",
      "iv": "WYUjVrJfeGGeXOha"
    }
  ]
}

// file_accessed
{
  "user_id": "abcdef0123456789...",
  "file_id": "9c3f3c3b30...",
  "filename": "informe-final.pdf",
```

```
}

// file_deleted
{
  "user_id": "abcdef0123456789...",
  "file_id": "9c3f3c3b305...f3efab04c6e372",
  "filename": "informe.pdf"
}

// file_renamed
{
  "user_id": "abcdef0123456789...",
  "file_id": "9c3f3c3b30...",
  "filename": "informe-final.pdf",
  "new_name": "informe_v2.pdf"
}

// file_replicated
{
  "file_id": "9c3f3c3b3...b04c6e372"
}
```

Anexo N. Formato de mensaje MQTT

Formato JSON utilizado para la transmisión de mensajes a través del canal MQTT. Estos mensajes actúan como notificaciones ligeras y permiten a los nodos conocer en tiempo real la publicación de nuevos eventos en IOTA. En la **Tabla 48** se detalla la estructura del mensaje y los campos necesarios para detectar, recuperar y procesar un evento completo.

Tabla 48. Estructura del mensaje MQTT usado para la sincronización entre nodos

Campo	Descripción
block_id	Identifica el evento a recuperar de IOTA.
event_type	Qué tipo de evento es: node_status, file_created, file_shared, ...
timestamp	Cuándo se emitió el evento. Facilita auditoría, ordenamiento, depuración, ...
node_id	Si el evento fue generado por un nodo concreto, quién lo emitió.

Fuente: elaboración propia.

Ejemplo:

```
{  
  "block_id": "0xabcdef1234567890...",  
  "event_type": "node_status",  
  "timestamp": "2025-04-25T20:45:00Z",  
  "node_id": "abcdef0123456789..."  
}
```

Anexo O. Fichero de configuración

El fichero de configuración `node.json` contiene la configuración local del nodo. Se genera automáticamente la primera vez que éste se inicia y almacena información esencial para su identificación y operación dentro de la red. La **Tabla 49** describe los campos que lo componen.

Tabla 49. Descripción de los campos de `node.json`

Campo	Descripción
<code>hostname</code>	Nombre de host o dirección pública del nodo (por ejemplo, un dominio DNS o IP). Se usa para la identificación en red.
<code>alias</code>	Nombre del nodo amigable asignado por el usuario. Usado para referencia humana.
<code>creation_date</code>	Fecha y hora de creación del fichero de configuración, en formato ISO.
<code>version</code>	Versión del software del nodo, útil para compatibilidad y diagnósticos.
<code>node_id</code>	Identificador único del nodo, calculado como el hash SHA-256 de su clave pública. Se utiliza como ID oficial en la red.
<code>port</code>	Puerto de red que expone el nodo para su interfaz HTTP.
<code>tags</code>	Etiquetas opcionales asociadas al nodo. Permiten clasificar o identificar características (ej. "test", "madrid").
<code>keys</code>	Contiene sus claves criptográficas y <i>salts</i> derivadas.

Fuente: elaboración propia.

Dentro del campo Keys, están las claves criptográficas y *salts*⁵⁶ derivadas, tal y como se indica en la **Tabla 50**. Las claves se generan, almacenan de forma segura y difunden (solo clave pública) para que el nodo se identifique en la red y firme los eventos que publica.

Tabla 50. Descripción del objeto `keys` en `node.json`

Campo	Descripción
<code>salt_private_key</code>	Sal aleatoria utilizada para衍生 determinísticamente la clave privada a partir de la <i>passphrase</i> introducida por el usuario.

⁵⁶ Una *salt* (o "sal" criptográfica) es un valor aleatorio que se añade a una entrada (por ejemplo, una contraseña) antes de aplicar una función de derivación de clave o hash. Su objetivo principal es evitar ataques pre-computados como las tablas arco iris (*rainbow tables*) al asegurar que la misma entrada no genera siempre el mismo resultado y dificultando los ataques por diccionario o fuerza bruta.

Campo	Descripción
salt_encryption	Sal para derivar la clave simétrica que se utiliza para cifrar la clave privada antes de almacenarla.
public_key	Clave pública del nodo, usada para verificar la firma de los eventos generados por el nodo.
private_key_encrypted	Clave privada, cifrada con una clave simétrica derivada de una <i>passphrase</i> y <i>salt_encryption</i> . Se descifra en memoria durante la inicialización y autenticación del nodo.

Fuente: elaboración propia.

Ejemplo:

```
{
  "node_id": "4e97faf4ec9fb5454a4b0a54b26ce6a4a682264e80f2b79ccab2d16e2870ac9f",
  "hostname": "nodo1.dfs3.net",
  "alias": "nodo-pruebas",
  "creation_date": "2025-05-02T06:16:29.260265+00:00",
  "version": "dfs3-node/0.1.0",
  "port": "3001",
  "tags": ["test", "madrid"],
  "keys": {
    "salt_private_key": "hEWEPAK0c6AgbGXTk/Lk0g==",
    "salt_encryption": "9cFTSp6KUYBgOZk8wsB1oQ==",
    "public_key": "R1u/z1W+Ee/F1Hpi+/GgXVY4KmIBl0mcAjOMCuimnc=",
    "private_key_encrypted": "..."
  }
}
```

Anexo P. Erasure Coding

En la arquitectura actual, los archivos cifrados se almacenan de forma completa en el nodo propietario y se distribuyen copias a varios nodos adicionales. Para aumentar la disponibilidad, tolerancia a fallos, flexibilidad y optimización de espacio, se plantea una mejora consistente en el uso de *Erasure Coding*. Esta replicación está inspirada en esquemas como *Reed-Solomon* y se utiliza en plataformas como *Amazon S3* o *IPFS*.

Principio de operación

El principio de operación basado en *Erasure Coding* permite distribuir de forma eficiente y resiliente los archivos. En lugar de replicar el fichero completo, este se divide en bloques. A su vez, cada bloque se divide en fragmentos codificados, de los cuales solo una parte es necesaria para reconstruir el fichero original. Esta técnica optimiza el uso del espacio y permite la recuperación fiable, incluso si varios nodos fallan o se pierden. El algoritmo propuesto es el siguiente:

1. Subida del archivo:

- El archivo se cifra localmente y se sube al nodo propietario.
- El nodo propietario publica un evento `file_created` en IOTA con sus metadatos (ID, esquema de fragmentación, hashes, tamaño, etc.).

2. Determinismo distribuido:

- Todos los nodos receptores que escuchan el evento `file_created` evalúan su elegibilidad (espacio libre, *uptime*, carga, etc.).
- Cada nodo ejecuta una función de hash determinista basada en el `file_id` y su propio `node_id` para calcular si debe participar en la replicación.
- Si resulta seleccionado entre los primeros $N=12$ nodos, determina su índice de fragmento asignado (ej. 3 → fragmento 3 de 12).
- Una alternativa a este esquema es que sea el propio nodo emisor el que seleccione quién participará de la replicación.

3. Descarga y fragmentación descentralizada:

- El nodo replicador descarga el archivo completo desde el nodo propietario mediante `GET /files/{file_id}/data`.
- Verifica el hash SHA-256 (`file_id`) para asegurar su integridad.
- Fragmenta localmente el archivo usando *Erasure Coding* ($K=8$, $M=4$) y extrae solo el fragmento que le corresponda.
- Almacena su fragmento local, descartando el resto.
- Una alternativa a este esquema es que el nodo replicador descargue únicamente su fragmento, implementando mayor inteligencia en el nodo propietario para la generación de fragmentos.

4. Publicación del evento `file_replicated`:

- El nodo replicador publica un evento `file_replicated` en IOTA incluyendo: `file_id`, índice del fragmento, hash del fragmento y nodo de origen.
- Esto permite al resto de nodos reconstruir el mapa de distribución de los fragmentos de ese fichero.

Esquema propuesto

El archivo cifrado se fragmenta utilizando un esquema $(K, M) = (8, 4)$:

- Se generan 8 fragmentos de datos (suficientes para reconstruir el archivo),
- Se generan 4 fragmentos adicionales de paridad, lo que permite tolerar hasta la pérdida de 4 fragmentos (nodos).
- 12 fragmentos en total, distribuidos entre nodos independientes.

Ventajas de la propuesta

- **Alta disponibilidad:** se tolera la pérdida de hasta M (4) nodos replicadores sin afectar a la capacidad de reconstrucción.
- **Resiliencia autónoma:** los nodos replicadores actúan de forma independiente y pueden reintentar la descarga si falla.

- **Verificabilidad:** cada nodo valida localmente el archivo cifrado antes de almacenar su fragmento.
- **Escalabilidad horizontal:** el sistema se adapta fácilmente al crecimiento de la red sin modificar la lógica de replicación.
- **Descentralización total** (opcional): no hay necesidad de coordinación explícita por parte del nodo propietario.

Consideraciones

- Que los nodos replicadores descarguen el archivo completo puede ser costoso en términos de ancho de banda, aunque se compensa con la simplificación del flujo. Al realizarse la fragmentación completamente en el nodo replicador, se requiere un menor uso de CPU (crítico en sistemas IoT).
- Esta implementación permite coexistencia con modelos de replicación directa en fases tempranas, con ficheros de poco tamaño o en escenarios de baja conectividad.
- Se puede crear una prueba de descarga: por ejemplo, el nodo replicador copia un fragmento y publica un evento que contiene el hash del segmento. El nodo propietario verifica y emite un evento de '*commit*' para que el resto de nodos actualicen su estado y lo den por válido.
- Como mejora, en el futuro, se puede implementar un mecanismo de '*pinning*' que permita descartar o no una copia local en función de lo crítico que consideremos el fichero replicado, similar a como hace IPFS.

Detalle de implementación

A continuación, se discute la implementación de *erasure coding* para una primera versión simplificada y con menos nodos ($K=3$, $M=2$).

Estructura de datos

Como propuesta de estructura de metadatos mostramos el siguiente ejemplo:

```
{  
  "k": 3,  
  "m": 2,  
  "block_size": 1572864,  
  "block_count": 3,  
  "blocks": [  
    {  
      "block_index": 0,  
      "fragments": [  
        { "fragment_index": 0, "node_id": "node01" },  
        { "fragment_index": 1, "node_id": "node02" },  
        { "fragment_index": 2, "node_id": "node03" },  
        { "fragment_index": 3, "node_id": "node04" },  
        { "fragment_index": 4, "node_id": "node05" }  
      ]  
    },  
    {  
      "block_index": 1,  
      "fragments": [...]  
    },  
    {  
      "block_index": 2,  
      "data_size": 1572864, // solo en el último si < block_size  
      "fragments": [...]  
    }  
  ]  
}
```

Detalles clave de la estructura:

- Los parámetros `k`, `m` se definen a nivel global.
- El valor del campo `block_size` es específico para cada fichero para minimizar el *padding*.
- Cada bloque incluye los siguientes campos:
 - `block_index`: número de bloque.
 - `data_size`: solo si es menor que `block_size` (tamaño del bloque final).
 - `fragments[]`: lista de fragmentos ya replicados. Cada `fragment` incluye los siguientes valores:
 - `fragment_index`: número de fragmento para ese bloque.
 - `node_id`: nodo que replica el fragmento.

Flujo de replicación

- Nodo emisor (receptor del archivo):

- Recibe el archivo completo y decide la asignación de fragmentos a nodos.
- Publica un evento `file_created` que incluye la asignación completa por bloque.
- Nodos replicadores:
 - Escuchan el evento `file_created`.
 - Determinan si deben replicar alguno de los fragmentos asignados. Si es así:
 - Solicitan su fragmento al nodo emisor mediante: GET `/files/{file_id}/block/{block_index}/fragment/{fragment_index}`
 - Guardan el fragmento localmente y publican un evento `file_replicated` con los siguientes campos:
 - `file_id`
 - `block_index`
 - `fragment_index`
 - `node_id`

Actualización de estado de replicación

- Cada nodo escucha los eventos `file_replicated`.
- Actualiza su `.meta/<file_id>.json`, añadiendo únicamente los fragmentos replicados con éxito. Esta estructura se construye y mantiene dinámicamente.

Limitaciones y consideraciones adicionales

En el esquema propuesto:

- La detección del estado de replicación es responsabilidad local de cada nodo.
- De momento no se tiene en cuenta el reintento si un nodo no replica el fragmento que le fue asignado.
- La estructura `.meta` refleja el estado actual real, no el plan teórico completo.

Estrategia inicial de asignación de fragmentos por índice fijo

Dado que el sistema está diseñado inicialmente para operar con un número reducido de nodos, se adopta una estrategia simple y eficiente para distribuir los fragmentos generados mediante codificación *Reed-Solomon*. En esta estrategia, cada nodo se responsabiliza de replicar siempre el mismo índice de fragmento en todos los bloques de un archivo.

Por ejemplo, si se codifica un archivo con $K=3$ y $M=2$ (total de 5 fragmentos por bloque) y hay cinco nodos disponibles (*node0* a *node4*), se asigna cada fragmento por índice a cada nodo. Así, *node1* almacenará siempre el fragmento 1 del bloque 0, del bloque 1, del bloque 2, etc.

Como ventajas de esta estrategia tenemos:

- Simplicidad en la asignación: solo se necesita emitir una única tabla de correspondencia `fragment_index → node_id` en el evento `file_created`.
- Reducción de metadatos: no es necesario detallar la asignación para cada bloque por separado.
- Carga equilibrada: si todos los bloques tienen igual tamaño, los nodos tienen la misma cantidad de fragmentos.
- Determinismo: permite a cada nodo saber de antemano qué fragmentos debe replicar, basándose solo en su `node_id`.

Anexo Q. Estrategia de replicación con incentivos económicos

Como evolución natural del sistema de compartición, se plantea una posible mejora que consiste en la incorporación de incentivos económicos al proceso de replicación y de acceso a ficheros. Ésta mejora, integra un mecanismo de selección determinista con un sistema de recompensas distribuido mediante tokens IOTA que aprovecha las capacidades de los contratos inteligentes en la red *Shimmer*.

Selección determinista de nodos replicadores

Para asegurar que la replicación no dependa del azar ni de decisiones centralizadas, se propone un algoritmo determinista con ponderación por mérito, que calcule el ranking por nodo candidato en función de valores como espacio disponible, *uptime*, historial de replicación exitosa o saldo de tokens acumulado (para evitar concentración). Una fórmula representativa podría ser la siguiente:

```
score_boost = (
    alpha * normalized(free_space) +
    beta * normalized(uptime) +
    gamma * normalized(past_reliability) -
    delta * normalized(wallet_balance) # Penaliza nodos saturados de tokens
)
ranking = (
    weight_random * (hash(file_id + node_id) % 1000) +
    weight_score * (max_score - score_boost)
)
```

Donde `file_id` y `node_id` son hashes SHA-256 y `score_boost` es un valor ponderado en el rango 0-100. La función `normalized(x)` escala entre [0, 1]. Por su parte, `alpha`, `beta`, `gamma` y `delta` son pesos configurables. Se puede añadir penalización si el nodo replicó recientemente, para diversificar. También se puede usar `weight_random` y `weight_score` para ajustar el peso relativo de mérito vs aleatoriedad. El valor `max_score` escala el score al mismo rango (ej. 100).

Una vez calculado, el nodo propietario selecciona los nodos con menor ranking para almacenar los fragmentos del fichero. En concreto, se seleccionan los $k + m$ primeros nodos para la replicación *Reed-Solomon*. Este sistema permite priorizar a nodos fiables sin eliminar del todo la variabilidad y puede ajustarse fácilmente modificando los pesos o el rango del hash.

Verificación de almacenamiento

Antes de liberar el pago por replicación, el nodo propietario puede ejecutar una prueba de almacenamiento ligera basada en el cálculo del hash SHA-256 del fragmento replicado:

- Por ejemplo, el nodo original envía un desafío indicando el `file_id`, `block_index` y `fragment_index` asignado al nodo replicador (puede ir en el evento `file_created`).
- El nodo replicador responde con el hash del fragmento que guarda localmente (puede ir en el evento `file_replicated`).
- Si el hash coincide con el valor esperado, el nodo propietario aprueba la verificación.

Integración con contratos inteligentes de IOTA (ISCP)

Para una gestión segura y descentralizada de los pagos, se propone el uso de IOTA *Smart Contracts* desplegados sobre nodos *Wasp*⁵⁷. En la **Tabla 51** se detallan las funciones principales que compondrían la propuesta de contrato inteligente para gestionar la replicación incentivada de archivos. Este contrato gestionaría:

- El depósito de tokens del usuario al subir un fichero.
- El registro de nodos seleccionados como replicadores.
- La validación de las réplicas mediante pruebas firmadas.
- La distribución automática de tokens a los nodos validados.

Tabla 51. Funciones del contrato inteligente propuestas para incentivo económico

Función	Descripción
<code>upload_file(file_id, nodes, total_tokens)</code>	Se llama al subir el fichero. Registra nodos y cantidad a repartir.
<code>verify_replica(file_id, node_id, hash)</code>	Llamada por el nodo replicador para demostrar que tiene el fragmento.

⁵⁷ *Wasp* es el nodo de ejecución de contratos inteligentes desarrollado por la IOTA Foundation. Permite desplegar y ejecutar *Smart Contracts* sobre una red descentralizada, utilizando la *Tangle* como capa de consenso y comunicación. A diferencia de los nodos tradicionales como *Hornet* (centrados en la red base), los nodos *Wasp* están diseñados específicamente para operar con cadenas de contratos inteligentes, permitiendo lógica programable y transacciones complejas de forma segura y escalable.

Función	Descripción
<code>release_payment(file_id, node_id)</code>	Interna. Si la verificación es correcta, transfiere tokens a ese nodo.
<code>get_status(file_id)</code>	Consulta para saber qué nodos han replicado ya.

Fuente: elaboración propia.

Estado del contrato (simplificado):

```
struct FileReplication {
    file_id: String,
    uploader: Address,
    total_tokens: u64,
    // Incluye node_id, fragment_index, estado
    node_list: [NodeInfo],
}
```

Como conclusión, este enfoque permitiría la construcción futura de un ecosistema autosostenible, automatizable y transparente, donde los nodos recibirían incentivos justos y los usuarios pagarían en función del uso.

Anexo R. Autenticación dual mediante JWT

Una mejora futura interesante consiste en la incorporación de un mecanismo de autenticación basado en tokens JWT (*JSON Web Token*). Este sistema permitiría establecer sesiones de usuario de forma segura y *stateless*, evitando la dependencia de sesiones almacenadas en memoria del servidor y mejorando la escalabilidad y compatibilidad con arquitecturas distribuidas.

La gestión de sesiones tradicionales (como tokens JWT firmados por el servidor) implica confiar en una autoridad emisora, algo incompatible con una red compuesta por nodos potencialmente heterogéneos y no plenamente confiables. Por otro lado, delegar por completo la autenticación al cliente (modelo de firma local) puede resultar demasiado laxo, al carecer de mecanismos de caducidad, trazabilidad o control.

Este anexo propone un modelo híbrido que **combina la soberanía criptográfica del usuario con un sistema de visado temporal por parte de un nodo**, de forma que se conserve la descentralización sin renunciar a una autenticación robusta, validable entre nodos.

Objetivo

El objetivo principal es sustituir el esquema actual basado en almacenamiento local de sesiones por una solución moderna en la que el usuario emita un token firmado digitalmente por él mismo y visado por el nodo *backend* que identifique al usuario durante un periodo de tiempo limitado.

Principios del modelo propuesto

1. Desafío firmado (*challenge*) para asegurar autenticidad en tiempo real.
2. El usuario firma su propio token de sesión, demostrando su identidad sin requerir de terceros.
3. Un nodo valida y “visa” ese token, añadiendo una firma adicional que lo convierte en una sesión temporal autorizada.

4. Otros nodos pueden validar la firma del usuario y el visado del nodo, sin necesidad de compartir claves secretas.

Flujo de autenticación

El proceso completo de autenticación quedaría estructurado en los siguientes pasos:

1. Obtención del desafío (challenge): el cliente web solicita al servidor un desafío aleatorio para su `user_id`. El servidor responde con un valor aleatorio (`nonce`) asociado al `user_id` y una validez temporal (por ejemplo, 60 segundos):

```
GET /auth/challenge?user_id=<id>
```

2. Generación del token local: el usuario genera localmente un objeto `user_token` y lo firma con su clave privada.

```
{  
  "user_id": "e638b7...",  
  "challenge": "N8hdjl19s8kc7Q9qKqU1zg==",  
  "issued_at": "2025-06-18T09:43:00Z",  
  "expires_in": 3600  
}
```

3. Petición de visado: el cliente envía `user_token + signature` al nodo mediante:

```
POST /auth/validate  
{  
  "user_token": { ... },  
  "signature": "<firma del user_token con la clave privada del usuario>"  
}
```

4. Visado del nodo: el nodo valida la firma del usuario, que el desafío coincide y que el `issue_at` es reciente. Si todo es correcto, devuelve el token visado con su propio `node_id`, fecha de expiración del visado y firma digital del conjunto anterior.

```
{  
  "user_token": { ... },  
  "signature": "<firma del usuario>",  
  "visa": {  
    "node_id": "4e97fa...870e",  
    "issued_at": "2025-06-18T09:43:05Z",  
    "expires_at": "2025-06-18T10:43:05Z",  
    "signature": "<firma del conjunto anterior con la clave privada del nodo>"  
  }  
}
```

5. Uso posterior: el cliente incluye el token completo como cabecera en sus peticiones:

Authorization: DFS3 user=<user_id>, token=<base64 del objeto completo>

6. Validación en nodos remotos: cualquier nodo receptor puede verificar:

- La firma del user_token con la clave pública del usuario.
- Que el challenge fue emitido por un nodo y que aún es válido.
- La firma del visado con la clave pública del nodo visador.
- Que ambas firmas son válidas y que las fechas no han expirado.

Estructura del JWT

El contenido del JWT, podría tener la siguiente forma:

```
{  
  "user_token": { ... },  
  "signature": "<firma del user_token con la clave privada del usuario>",  
  "visa": {  
    "node_id": "4e97fa...870e",  
    "expires_at": "2025-06-18T10:42:00Z",  
    "signature": "<firma del conjunto anterior con la clave privada del nodo>"  
  }  
}
```

Ventajas

- Escalabilidad: no requiere almacenar sesiones en memoria.
- Compatibilidad distribuida: cualquier nodo puede validar el token.
- Simplicidad: se aprovechan estándares ampliamente soportados.
- Seguridad:
 - Los tokens son auto-verificables y caducan automáticamente.
 - El desafío garantiza que el token no puede ser reutilizado ni clonado.
 - El nodo puede controlar la duración del visado y el uso del desafío.
 - Solo un usuario con la clave privada y en posesión del desafío actual puede autenticarse.

Conclusión

Este modelo conserva las ventajas de un enfoque híbrido (soberanía del usuario, control temporal por parte del nodo y validación distribuida) y añade la seguridad necesaria para evitar ataques de repetición o suplantación.

Consideraciones adicionales

- Podría incluirse un sistema de revocación (*blacklist de tokens*) para casos de cierre de sesión o robo de credenciales.
- También sería viable cifrar el contenido del JWT (JWE) si se desea proteger la información frente a intermediarios.
- Sería especialmente útil si el sistema evoluciona hacia una arquitectura federada o multi-nodo donde distintos nodos pueden validar peticiones.

Anexo S. VPS de desarrollo y pruebas

Con el objetivo de garantizar un entorno estable, accesible para el desarrollo y validación del sistema, se ha adquirido un servidor VPS dedicado. Este servidor ha sido esencial para desplegar y mantener los servicios base del sistema, permitiendo realizar pruebas en condiciones similares a las de un entorno real distribuido.

Listado de servicios

Su configuración incluye:

- **Nodo IOTA (Hornet)**: desplegado mediante Docker, actúa como capa de persistencia descentralizada para los eventos del sistema. Este nodo se ha mantenido en funcionamiento continuo para simular una red operativa y recibir las publicaciones generadas por los nodos dfs3.
- **Bróker MQTT (Mosquitto)**: configurado como canal de control, ha servido para distribuir en tiempo real los identificadores de eventos (`block_id's`) entre los nodos del sistema, facilitando la sincronización y propagación eficiente de cambios.
- **Servidor DNS dinámico**: permite asignar subdominios personalizados a cada nodo (por ejemplo, `nodo1.dfs3.net`, `nodo2.dfs3.net`, etc.), facilitando el acceso remoto mediante HTTPS y simplificando el enrutamiento desde el cliente web.
- **Nodo Cero (node0.dfs3.net)**: este nodo ha cumplido múltiples funciones estratégicas dentro del desarrollo:
 - **Nodo semilla**: punto de arranque para la propagación inicial de eventos y sincronización de nuevos nodos en la red.
 - **Nodo de pruebas**: entorno principal para validar funcionalidades, detectar errores y simular distintos escenarios de carga.
 - **Frontend web**: opcionalmente puede actuar como servidor de contenido estático para servir la interfaz gráfica de usuario desde <https://nodo.dfs3.net>, que permite seleccionar dinámicamente el nodo que actuará como *backend*, centralizando y simplificando la experiencia de usuario.

Figura 47. Selección de nodo backend en dfs3



Fuente: elaboración propia.

La **Figura 47** muestra la pantalla de selección de nodo en la interfaz web de la aplicación. Aparece cuando accedemos a través del nodo cero (<https://nodo.dfs3.net>). El usuario elige manualmente el nodo que desea usar como *backend* (idealmente, el suyo), esto permite distribuir la carga entre nodos y favorece la descentralización.

Al servir el *frontend* de la aplicación, hay que tener en cuenta que la interfaz web es de apenas 250Kb de contenido estático cacheable. El consumo de recursos necesarios para servir a cientos e incluso miles de usuarios es mínimo.

Características Hardware

El VPS utilizado se ejecuta sobre una máquina virtual con el perfil de hardware que aparece en la **Tabla 52**:

Tabla 52. Características HW del VPS

Parámetro	Valor
CPU	Intel(R) Xeon(R) E5-2670 @ 2.60GHz (1 vCPU asignado)
Memoria RAM	1 GB
Almacenamiento	60 GB SSD
Sistema operativo	Ubuntu Server 22.04 LTS

Fuente: elaboración propia.

A pesar de sus recursos limitados, el VPS ha demostrado ser suficiente para las tareas de desarrollo, pruebas y despliegue inicial. Esto demuestra la viabilidad de la solución en entornos de bajo coste con consumo moderado.

Configuración Nginx

En el servidor VPS se ha configurado un proxy inverso *Nginx* que actúa como punto de entrada único al sistema. Este componente cumple la función de *frontend*, repartiendo el tráfico entrante entre los distintos servicios desplegados en el servidor (API REST, *frontend* web, nodo IOTA, bróker MQTT) en función del subdominio o la ruta solicitada. Gracias a esta configuración, se simplifica el acceso desde el exterior y se garantiza una gestión centralizada de certificados TLS, autenticación y redirecciones, mejorando tanto la seguridad como la mantenibilidad del entorno.

A continuación, un extracto de la configuración de *Nginx*, usado como proxy inverso:

```
# ---
# http://iota.dfs3.net -> https://iota.dfs3.net
# ---
server {
    listen 80;
    server_name iota.dfs3.net;

    # necesario para validación certbot
    location /.well-known/acme-challenge/ {
        root /var/www/html/;
    }

    # redirigir el resto de tráfico a HTTPS
    location / {
        return 301 https://$host$request_uri;
    }
}

# ---
# https://iota.dfs3.net -> api IOTA en Hornet
# ---
server {
    listen 443 ssl;
    server_name iota.dfs3.net;

    # habrá que cambiarlo, de momento para pruebas
    ssl_certificate /etc/letsencrypt/live/iota.dfs3.net/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/iota.dfs3.net/privkey.pem;

    # nginx:443 -> traefik:8080 -> hornet:80
    location /api/core/v2 {
        proxy_pass http://localhost:8080;
    }
}

# ---
# http://node.dfs3.net -> https://node.dfs3.net
# ---
```

```
server {
    listen 80;
    server_name node.dfs3.net;

    # necesario para validación certbot
    location /.well-known/acme-challenge/ {
        root /var/www/html/;
    }

    # redirigir el resto de tráfico a HTTPS
    location / {
        return 301 https://$host$request_uri;
    }
}

# ---
# https://node.dfs3.net -> simplifica el acceso público a la interfaz de dfs3
# ---
server {
    listen 443 ssl;
    server_name node.dfs3.net;

    # Para limitar el tamaño de fichero a subir
    client_max_body_size 100M;

    # Configuración del certificado TLS generado con certbot
    ssl_certificate /etc/letsencrypt/live/node.dfs3.net/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/node.dfs3.net/privkey.pem;

    # nginx:443 -> dfs3:3000
    location / {
        proxy_set_header Host $host;
        proxy_pass https://localhost:3000;
    }
}

# ---
# http://mqtt.dfs3.net -> para validación de dominio
# ---
server {
    listen 80;
    server_name mqtt.dfs3.net;

    # necesario para validación certbot
    location /.well-known/acme-challenge/ {
        root /var/www/html/;
    }

    # prohibido el resto de tráfico http
    location / {
        return 404;
    }
}
```

Anexo T. Coste estimado del proyecto

Con el objetivo de valorar la viabilidad económica del sistema desarrollado, se ha realizado una estimación aproximada de los costes reales asociados al despliegue y pruebas del prototipo. Los componentes seleccionados reflejan una apuesta por soluciones accesibles y de bajo coste, en coherencia con el enfoque del proyecto. En la **Tabla 53** se detallan los principales conceptos involucrados, sin incluir tiempo de desarrollo ni recursos previamente disponibles:

Tabla 53. Desglose del coste estimado del proyecto

Concepto	Unidades	Precio unitario (€)	Coste total (€)
Dominio dfs3.net (1 año)	1	1.5	1.5
Servidor VPS (3 meses)	3	2.0	6.0
Tarjetas SD (4x32GB)	4	6.0	24.0
Orange Pi One (4 unidades)	4	12.0	48.0
Cables, fuentes y accesorios	-	5.0	5.0
Total	-	-	142.5 €

Fuente: elaboración propia.

Anexo U. Impacto social, económico y medioambiental

Aunque el proyecto se ha concebido como una solución técnica para el almacenamiento distribuido, su diseño y principios están fuertemente alineados con los valores sociales, económicos y medioambientales. Este anexo presenta una reflexión sobre el impacto positivo que podría tener su adopción generalizada, especialmente si se aplicara en comunidades o en redes colaborativas.

Impacto social

El sistema promueve un modelo de red descentralizada en el que cualquier usuario puede convertirse en un nodo activo del ecosistema y contribuir con su infraestructura personal a la red. Esta descentralización tiene implicaciones sociales relevantes:

- Soberanía de los datos: los usuarios mantienen el control sobre su información, sin depender de terceros centralizados.
- Acceso equitativo: se eliminan barreras económicas y técnicas para acceder a servicios de almacenamiento seguro.
- Colaboración distribuida: se facilita la creación de redes de confianza entre iguales, especialmente útil en entornos educativos o de investigación.

Impacto económico

Permite imaginar un modelo económico basado en el uso compartido de recursos y compensaciones por servicios prestados. El enfoque se alinea con una visión de economía circular y colaborativa, donde los recursos son aprovechados colectivamente:

- Incentivos a pequeños proveedores o particulares: personas con nodos activos podrían recibir micro-compensaciones por almacenar fragmentos cifrados, validar eventos o servir archivos.
- Reducción de costes de almacenamiento: al no depender de grandes proveedores comerciales, las organizaciones pequeñas y los proyectos colaborativos podrían desplegar soluciones de almacenamiento resiliente a bajo coste.

- Fomento del emprendimiento local: se podrían desarrollar servicios para sectores como la educación, investigación o incluso la administración local, sin necesidad de grandes inversiones iniciales.

Impacto medioambiental

El sistema se ha diseñado pensando en la eficiencia energética y la reutilización de infraestructura existente, lo que conlleva beneficios medioambientales tangibles:

- Uso de hardware de bajo consumo: dispositivos como Orange Pi, Raspberry Pi u otros pequeños servidores personales requieren de muy poca energía en comparación con los centros de datos tradicionales.
- Descentralización de la carga energética: al distribuir el almacenamiento y el procesamiento, se evita la concentración de consumo en grandes servidores o regiones específicas.
- Reutilización de equipos infráutilizados: muchos hogares y pequeñas organizaciones disponen de dispositivos capaces de participar, alargando su vida útil y reduciendo la generación de residuos electrónicos.

Conclusión

La arquitectura descentralizada y ligera del sistema propuesto abre la puerta a modelos de uso con impacto social, económico y medioambiental. Promueve la soberanía digital, democratiza el acceso al almacenamiento distribuido y ofrece una alternativa sostenible al modelo actual. En definitiva, puede considerarse no solo una propuesta funcional, sino también una herramienta alineada con los principios éticos y sostenibles de la era digital.

Índice de acrónimos

ABCI. <i>Application Blockchain Interface</i>	IPFS. <i>InterPlanetary File System</i>
AJAX. <i>Asynchronous JavaScript and XML</i>	LoRa. <i>Long Range</i>
API. <i>Application Programming Interface</i>	LoRaWAN. <i>Long Range Wide Area Network</i>
BFT. <i>Byzantine Fault Tolerance</i>	LPoS. <i>Leased Proof of Stake</i>
BLE. <i>Bluetooth Low Energy</i>	LPWAN. <i>Low-Power Wide-Area Network</i>
BTC. <i>Bitcoin</i>	LTE-M. <i>Long Term Evolution for Machines</i>
CoAP. <i>Constrained Application Protocol</i>	M2M. <i>Machine to Machine</i>
CSS. <i>Cascading Style Sheets</i>	MQTT. <i>Message Queuing Telemetry Transport</i>
DAG. <i>Directed Acyclic Graph</i>	NB-IoT. <i>Narrowband IoT</i>
DAOs. <i>Decentralized Autonomous Organizations</i>	NFT. <i>Non-Fungible Token</i>
DAPP. <i>Decentralized Application</i>	P2P. <i>Peer-to-peer</i>
DeFi. <i>Decentralized Finance</i>	PBFT. <i>Practical Byzantine Fault Tolerance</i>
DHT. <i>Distributed Hash Tables</i>	PoA. <i>Proof of Authority, Proof of Authority</i>
DLT. <i>Distributed Ledger Technology</i>	PoB. <i>Proof of Burn</i>
DPoS. <i>Delegated Proof of Stake</i>	PoC. <i>Proof-of-Coverage</i>
DSCSA. <i>Drug Supply Chain Security Act</i>	PoET. <i>Proof of Elapsed Time</i>
ESP32. <i>Espressif Systems 32-bit Microcontroller</i>	PoH. <i>Proof of History</i>
ETH. <i>Ethereum</i>	PoS. <i>Proof of Service</i>
EVM. <i>Ethereum Virtual Machine</i>	SBC. <i>Single Board Computer</i>
FDA. <i>Food and Drug Administration</i>	SDK. <i>Software Development Kit</i>
GxP. <i>Good x Practice</i>	SLAC. <i>Stanford Linear Accelerator Center</i>
IAG. <i>Iagon</i>	SSI. <i>Self-Sovereign Identity</i>
IoT. <i>Internet of Things</i>	STM32. <i>STMicroelectronics 32-bit Microcontroller</i>