Informe de Trabajo Profesional

Análisis de ecosistemas para la implementación de plataformas como servicio para despliegue de aplicaciones comunitarias, distribuidas y descentralizadas

Integrantes

Sebastian Bento Inneo Veiga
100998
sinneo@fi.uba.ar

Joaquín Prada
105978
jprada@fi.uba.ar

Lucas Nahuel Sotelo Guerreño
102730
lsotelo@fi.uba.ar

Joaquín Matías Velazquez
105980
jvelazquez@fi.uba.ar

Tutor
Ing. Ariel Scarpinelli
ascarpinelli@fi.uba.ar

Índice

1. Resumen **5**

2. Palabras Clave **5**

3. Abstract **5**

4. Keywords **5**

5. Agradecimientos **6**

6. Introducción **7**

7. Estado del Arte **7**

7.1. Introducción a arquitecturas de red . . . 7
7.2. Diferencias y ventajas de cada arquitectura . . . 8
7.3. Ambientes y herramientas . . . 10
7.3.1. IPFS . . . 10
7.3.2. Blockchain . . . 11
7.3.3. Alternativas . . . 12
7.4. Soluciones existentes . . . 12
7.4.1. IPFS Deploy Action . . . 12
7.4.2. Distributed Wikipedia Mirror . . . 13

8. Problema detectado y/o faltante **13**

8.1. Costo y sostenibilidad . . . 13
8.2. Interrupciones e infraestructura crítica . . . 13
8.3. Centralización y control de acceso . . . 13
8.4. Accesibilidad tecnológica . . . 13

9. Solución implementada **14**

9.1. Casos de uso . . . 14
9.1.1. Sitio web estático . . . 14
9.1.2. Repositorio de conocimiento . . . 14
9.1.3. Mensajero en tiempo real . . . 15
9.2. IPFS . . . 16
9.2.1. Infraestructura de despliegue . . . 16
9.2.2. Infraestructura de aplicación . . . 23
9.2.3. Colaboración . . . 31
9.3. Blockchain . . . 32
9.3.1. Swarm . . . 32
9.3.2. Ethereum . . . 33
9.4. Hyphanet . . . 35
9.4.1. Arquitectura . . . 35

9.4.2. Plugins . . . 35
9.4.3. Sitio web estático . . . 36
9.4.4. Otros casos de uso . . . 36
9.5. Freenet . . . 36
9.5.1. Arquitectura . . . 36
9.5.2. Diferencias con Hyphanet . . . 37
9.5.3. Baja del ecosistema . . . 37
9.6. Interfaces de usuario . . . 37

10.Metodología aplicada **42**

11.Experimentación y/o validación **43**
11.1. Propiedades utilizadas para las métricas . . . 43
11.1.1. Sitio Web Estático . . . 43
11.1.2. Repositorio de conocimiento . . . 43
11.1.3. Mensajero en tiempo real . . . 44
11.2. IPFS . . . 44
11.2.1. Costos . . . 44
11.2.2. Experiencia de desarrollo . . . 44
11.2.3. Aplicabilidad al caso de uso . . . 45
11.2.4. Performance . . . 45
11.2.5. Sistema utilizado . . . 51
11.3. Blockchain . . . 51
11.3.1. Costos . . . 51
11.3.2. Experiencia de desarrollo . . . 52
11.3.3. Aplicabilidad al caso de uso . . . 52
11.3.4. Performance . . . 53
11.3.5. Sistema utilizado . . . 56
11.4. Resumen . . . 56

12.Cronograma de las actividades realizadas **57**

13.Riesgos materializados **59**

14.Lecciones aprendidas **60**
14.1. Tecnologías emergentes . . . 60
14.2. Creación de paquetes . . . 60
14.3. Otras lecciones . . . 60

15.Impactos sociales y ambientales **60**
15.1. Aplicaciones en IPFS . . . 61
15.1.1. Moderación y Censura . . . 61
15.1.2. Impacto ambiental . . . 62
15.2. Aplicaciones en Ethereum . . . 62

15.2.1. Moderación y Censura 62
15.2.2. Impacto ambiental 63

16. Trabajos futuros **63**
16.1. Mejoras a los casos de uso 63
16.1.1. AstraDB 63
16.1.2. Astrawiki-eth 64
16.1.3. Astrachat-eth 64
16.2. Mejora para clusters colaborativos 64
16.3. Análisis del consumo de energía en la red de IPFS 65
16.4. Blockchain para aplicaciones comunitarias 65
16.5. Análisis de Freenet como ecosistema 65
16.6. IPFS + Ethereum 65

17. Conclusiones **66**

18. Referencias **66**

19. Anexos **69**
19.1. Arquitectura del repositorio de conocimiento 69
19.1.1. Representación de un artículo 69
19.1.2. Concurrencia 70
19.1.3. Árbol de versiones 70
19.1.4. Arquitectura del mensajero en tiempo real 72
19.2. Elección de herramientas en la arquitectura de despliegue de IPFS 73
19.2.1. Persistencia 73
19.2.2. Acceso y mutabilidad 73
19.3. Selección del protocolo de transporte para conexiones web en IPFS 74
19.4. Listado de issues reportados en herramientas del ecosistema de IPFS 75
19.5. Repositorios 76

1. Resumen

En el presente trabajo se analizan los ecosistemas de IPFS, Ethereum y Hyphanet/Freenet, junto con sus herramientas asociadas, con el objetivo de evaluar su uso en el desarrollo y despliegue de aplicaciones comunitarias de manera distribuida y descentralizada.

Esto se logra mediante la implementación de tres casos de uso que ilustran diferentes requisitos: un sitio web estático, un repositorio de conocimiento y un mensajero en tiempo real, a partir de los cuales se realiza un análisis cualitativo y cuantitativo de cada ecosistema.

2. Palabras Clave

Comunitario. Distribuido. Descentralizado. IPFS. Blockchain. Ethereum. Sistema. Aplicación. Peer-to-peer. Hyphanet. Freenet.

3. Abstract

The present work analyzes the ecosystems of IPFS, Ethereum, and Hyphanet/Freenet, along with their associated tools, with the aim of evaluating their use in the development and deployment of community applications in a distributed and decentralized manner.

This is achieved through the implementation of three use cases that illustrate different requirements: a static website, a knowledge repository, and a real-time messenger, from which a qualitative and quantitative analysis of each ecosystem is conducted.

4. Keywords

Community. Distributed. Decentralized. IPFS. Blockchain. Ethereum. System. Application. Peer-to-peer. Hyphanet. Freenet.

5. Agradecimientos

En primer lugar, queremos agradecer a la Facultad de Ingeniería de la Universidad de Buenos Aires y a todos los docentes que contribuyeron a nuestra formación académica y profesional por la educación de calidad que recibimos a lo largo de estos años.

Extendemos un agradecimiento a la comunidad *open source* de OrbitDB y Libp2p. Su trabajo y disposición para ayudar fueron fundamentales para la concreción de este proyecto.

Finalmente, queremos agradecer a todas las personas que nos acompañaron en este trayecto, incluyendo a nuestras familias y amigos, por el afecto, paciencia, comprensión y el apoyo que nos brindaron en el transcurso de nuestra carrera universitaria.

6. Introducción

El presente trabajo tiene como objetivo explorar alternativas de infraestructura para el desarrollo y despliegue de aplicaciones comunitarias, descentralizadas y distribuidas. A diferencia de los servicios centralizados convencionales, este tipo de aplicaciones enfrentan desafíos particulares que no siempre pueden resolverse de manera eficaz mediante las plataformas tradicionales.

En la actualidad, la mayoría de los servicios web se apoyan en proveedores de infraestructura como AWS, Azure o Google Cloud, que ofrecen soluciones ágiles para el desarrollo, despliegue y mantenimiento de aplicaciones. Sin embargo, este modelo no siempre resulta adecuado para aplicaciones comunitarias, entendidas como aquellas creadas, mantenidas y/o gobernadas por una comunidad. En estos casos, la toma de decisiones está distribuida entre los participantes, orientada al bien común y basada en la colaboración entre usuarios. Esta estructura permite que la evolución de la aplicación responda directamente a las necesidades de su comunidad.

Las aplicaciones comunitarias suelen operar con presupuestos limitados, sustentándose a través de voluntariado o donaciones, y pueden verse afectadas por políticas de censura u otras restricciones en determinadas regiones. Además, su propósito suele estar alineado con valores como la apertura y el acceso equitativo a la información, lo que entra en tensión con las lógicas de control y dependencia que caracterizan a las infraestructuras centralizadas.

En este contexto, surgen ecosistemas alternativos —basados en tecnologías *peer-to-peer*— que permiten a las aplicaciones ser alojadas y mantenidas directamente por sus usuarios, compartiendo recursos como almacenamiento y conectividad. Este enfoque distribuye la infraestructura entre múltiples nodos, eliminando puntos únicos de fallo, aumentando la resistencia a la censura y reduciendo los costos operativos.

Con el objetivo de evaluar estas tecnologías, se diseñaron e implementaron diversos casos de uso representativos, como un sitio web estático, un repositorio de conocimiento y un mensajero en tiempo real. Estos casos permitieron analizar la viabilidad técnica y operativa de construir aplicaciones comunitarias completamente descentralizadas, sin depender de infraestructura de terceros.

Este documento presenta un análisis detallado de la infraestructura disponible para la implementación y el despliegue de aplicaciones comunitarias descentralizadas. A través de casos de uso concretos, se examinan las capacidades y restricciones de cada ecosistema evaluado, destacando su aplicabilidad y potencial que ofrece este enfoque alternativo.

7. Estado del Arte

En esta sección describiremos en qué se diferencian las aplicaciones descentralizadas de aquellas centralizadas, cuáles son las ventajas (y desventajas) del modelo de aplicación distribuido, y qué tecnologías existen actualmente para asistir en la creación de dichas aplicaciones.

7.1. Introducción a arquitecturas de red

La comunicación entre distintas computadoras requiere una coordinación efectiva entre todas las partes involucradas, el uso de protocolos que estandaricen la forma en que se transmite la información, y una infraestructura que permita enviar cada *bit* desde el origen hasta su destino. Para ofrecer servicios a través de Internet, es necesario diseñar una red bien estructurada que cumpla con estos requisitos. En este contexto, existen dos arquitecturas principales que permiten alcanzar dicho objetivo.

Cliente-Servidor

El modelo Cliente-Servidor es ampliamente utilizado en la mayoría de las aplicaciones disponibles en Internet. Consiste en un nodo central, el servidor, responsable de gestionar las interacciones entre los usuarios, así como las comunicaciones directas entre cada usuario y el propio servidor. En

este esquema, los demás nodos actúan como clientes, solicitando servicios o recursos que el servidor proporciona.

Este modelo se clasifica como una arquitectura centralizada, ya que la subred depende directamente del servidor. En caso de que este nodo central falle, los clientes pierden la capacidad de comunicarse entre sí o acceder a los servicios, lo que representa un punto único de fallo.

Entre los servicios más comunes que adoptan esta arquitectura se encuentran la World Wide Web (HTTP/HTTPS), el correo electrónico (SMTP, IMAP) y el sistema de DNS, entre otros.

Peer-to-Peer

El modelo ***peer-to-peer***(P2P) consiste en una red **descentralizada** compuesta de diferentes nodos capaces de comunicarse sin necesidad de un nodo central, por lo que se puede considerar que cada nodo cumple la función tanto de servidor como de cliente.

BitTorrent El servicio más utilizado que implementa este modelo es la red de BitTorrent [57], que implementa el protocolo del mismo nombre para compartir archivos entre pares. Esta red logra que el mismo nodo que descarga un contenido de la red sea a la vez el servidor para otro nodo que quiera acceder a ese contenido.

7.2. Diferencias y ventajas de cada arquitectura

Ambos modelos tienen ventajas y desventajas, y por lo tanto distintos casos de uso. El modelo cliente-servidor actualmente es la arquitectura más utilizada.

Resiliencia En el modelo cliente-servidor, la disponibilidad de la red depende críticamente del servidor central. Si este falla o queda fuera de servicio, los clientes no pueden acceder a los recursos, lo que puede interrumpir por completo el funcionamiento del sistema. Para mitigar esta vulnerabilidad, se han implementado soluciones como la replicación de servidores en distintas ubicaciones geográficas, el balanceo de carga [18] y el uso de fuentes de energía alternativas, todo con el fin de mejorar la tolerancia a fallos y garantizar una mayor disponibilidad.

En contraste, una red P2P se basa en una arquitectura descentralizada en la que los nodos actúan simultáneamente como clientes y servidores. Esta estructura distribuye la responsabilidad entre múltiples participantes, lo que permite que la red siga operando incluso si varios nodos se desconectan. Cuando los pares están lo suficientemente dispersos geográficamente y hay una cantidad adecuada de ellos, el sistema puede mantener su funcionalidad frente a fallos locales, cortes de energía o incluso desastres naturales. Esta resiliencia inherente hace que el modelo P2P sea especialmente atractivo para servicios que requieren alta disponibilidad o continuidad operativa en condiciones adversas.

No obstante, la robustez de una red P2P depende del nivel de redundancia de los datos distribuidos. Si la red cuenta con pocos nodos o si los datos no están replicados entre varios pares, la desconexión de un nodo puede provocar la pérdida temporal de parte del contenido. En estos casos, aunque la red permanezca activa, su funcionalidad puede verse comprometida. Por tanto, garantizar la resiliencia en un entorno P2P también implica diseñar mecanismos eficientes de replicación y distribución de la información.

Escalabilidad Una de las principales ventajas de las redes *peer-to-peer* (P2P) es su escalabilidad. A medida que se suman más nodos a la red, también aumentan los recursos disponibles —como capacidad de procesamiento, almacenamiento y ancho de banda— lo que permite distribuir la carga de forma más eficiente. En una red P2P bien diseñada, cada nuevo participante no solo consume recursos, sino que también contribuye al sistema, favoreciendo un crecimiento orgánico y sostenible.

En cambio, en el modelo cliente-servidor, escalar implica aumentar la capacidad del servidor o incorporar servidores adicionales para atender la creciente demanda. Esto conlleva costos signifi-

cativos en términos de infraestructura, mantenimiento y administración. En aplicaciones de gran escala, como redes sociales o servicios de *streaming*, es común la implementación de clústeres de servidores y redes de de CDN (*Content Delivery Network*) para evitar cuellos de botella y garantizar el rendimiento, lo cual aumenta la complejidad técnica y económica del sistema.

Control del contenido En una arquitectura cliente-servidor, el servidor central debe gestionar múltiples conexiones simultáneas, lo cual requiere una infraestructura robusta que la mayoría de los usuarios no está en condiciones de mantener. Por esta razón, muchas aplicaciones recurren al alojamiento en plataformas de computación en la nube, como AWS, Azure o Google Cloud. Si bien estos servicios ofrecen alta disponibilidad y escalabilidad, también centralizan el control del contenido. Esto implica que las plataformas de *hosting* tienen la capacidad de modificar, censurar o eliminar aplicaciones y servicios, ya sea por decisión propia, por presión de gobiernos, o en cumplimiento de políticas internas.

En contraste, una red *peer-to-peer* (P2P) distribuye el contenido entre los propios usuarios, quienes actúan como anfitriones del sistema. Esta descentralización hace que la moderación de contenido sea mucho más difícil, ya que no existe un único punto de control. Esta característica resulta valiosa en contextos donde el acceso a la información está restringido por censura gubernamental o limitaciones regulatorias. Sin embargo, también plantea desafíos éticos y legales, ya que dificulta el control sobre la difusión de contenidos ilícitos o perjudiciales. Así, el modelo P2P ofrece mayor libertad, pero también exige nuevas formas de regulación y responsabilidad colectiva.

Seguridad La seguridad en aplicaciones basadas en el modelo cliente-servidor ha sido ampliamente estudiada debido a su larga trayectoria y adopción masiva. La centralización permite al administrador del sistema aplicar políticas de seguridad de forma coherente, como el bloqueo de conexiones sospechosas, la autenticación de usuarios y la eliminación de contenido malicioso, todo sin requerir intervención directa del usuario final.

En cambio, las redes P2P presentan desafíos distintos. Al no existir un punto de control central, la responsabilidad de establecer conexiones seguras recae en cada nodo participante. La ausencia de una autoridad central puede dificultar la detección y mitigación de comportamientos maliciosos dentro de la red.

Sin embargo, independientemente del modelo arquitectónico adoptado, la seguridad general de una aplicación dependerá en gran medida de su propio diseño: el uso adecuado de cifrado, autenticación, y la gestión de vulnerabilidades son factores determinantes para garantizar la protección de los datos y de los usuarios.

Persistencia En redes descentralizadas, la persistencia depende del comportamiento de los nodos y de las características específicas de la aplicación. En sistemas como BitTorrent, cada nodo que descarga un archivo puede continuar compartiéndolo con otros participantes, lo que genera una forma de redundancia dinámica: a medida que más usuarios descargan un archivo, aumentan las copias disponibles en la red. Sin embargo, si un archivo no es ampliamente compartido, puede volverse inaccesible cuando los pocos nodos que lo contienen se desconectan. En este sentido, la persistencia en redes *peer-to-peer* no está garantizada por diseño, sino que depende del nivel de replicación voluntaria entre los pares.

Latencia Dada una conexión a Internet promedio, las velocidades manejadas por las aplicaciones cliente-servidor suelen ser aceptables. Sin embargo, en zonas en donde la conexión es escasa, o en casos en donde el servidor está lejos del cliente, la velocidad de transferencia de la aplicación puede verse afectada. Además, no es infrecuente encontrar cortes en videollamadas, videojuegos, y demás aplicaciones de tiempo real que siguen esta arquitectura. Como en la mayoría de defectos del modelo cliente-servidor, se puede solucionar agregando múltiples instancias del servidor. Por ejemplo, es común almacenar películas, videos y demás contenido de aplicaciones de *streaming* en distintos servidores de CDN. Estas redes minimizan la distancia entre el usuario y el servidor, agilizando así la transferencia del contenido.

A pesar de los avances en la optimización del modelo cliente-servidor, las redes descentralizadas, cuando son eficientes y están bien pobladas, suelen ofrecer incluso mejores resultados. Esto se debe a que la fuente de un contenido puede estar presente en múltiples nodos, lo que aumenta la probabilidad de que un nodo cercano tenga el contenido solicitado. La velocidad de transferencia que puede proporcionar un vecino con el contenido que requerimos generalmente superará la ofrecida por un servidor.

Costos Una de las ventajas del modelo *peer-to-peer* (P2P) es su bajo costo operativo. Dado que los propios usuarios de la red aportan recursos como almacenamiento, procesamiento y ancho de banda, no es necesario mantener una infraestructura centralizada para alojar la aplicación. Esto reduce considerablemente los gastos asociados a la implementación y mantenimiento del sistema.

En contraste, el modelo cliente-servidor requiere disponer de un servidor dedicado o contratar servicios de alojamiento web, cuyos costos suelen incrementarse a medida que la aplicación escala y atrae más usuarios. Estos gastos incluyen no solo el *hardware* o el servicio de *hosting*, sino también aspectos como soporte técnico, ancho de banda adicional y medidas de seguridad. Por ello, las arquitecturas descentralizadas pueden representar una alternativa económicamente viable, especialmente en contextos donde los recursos financieros son limitados o se busca una solución escalable con bajo mantenimiento central.

7.3. Ambientes y herramientas

Existen varios ecosistemas que apuntan a proveer un marco con el cuál desarrollar una aplicación descentralizada. A su vez, cada uno de ellos cuenta con herramientas especializadas para los diferentes tipos de aplicaciones.

7.3.1. IPFS

IPFS (*InterPlanetary File System*) es un conjunto modular de protocolos diseñado para la organización y transferencia de datos en una red *peer-to-peer*, basado en el principio de **direccionamiento por contenido**, es decir, la recuperación de archivos en función de su contenido y no de su ubicación o identificador arbitrario [15]. Su principal propósito es facilitar la publicación de datos como archivos, directorios y sitios web de forma descentralizada, conformando así un sistema de archivos global, distribuido y descentralizado.

Este enfoque representa una alternativa al modelo tradicional de la web, que se basa en el direccionamiento por ubicación (*location-based addressing*), como ocurre con HTTP. Dicho modelo impone limitaciones estructurales que son contrarias a los principios de descentralización y autonomía que caracterizan a las aplicaciones comunitarias y distribuidas.

IPFS, en cambio, propone una red abierta, participativa y sin control centralizado, donde cualquier usuario puede contribuir y operar como nodo. Esta descentralización brinda la posibilidad de evitar la dependencia de servicios de terceros para el despliegue de aplicaciones —lo cual puede resultar costoso—, al permitir que los propios usuarios colaboren activamente con recursos de sus dispositivos para sostener el funcionamiento de la red.

Además, gracias al direccionamiento por contenido, cada archivo en IPFS cuenta con un identificador único —llamado *Content Identifier*, o CID—, lo que permite un acceso persistente y distribuido desde múltiples ubicaciones. Esta característica mejora la disponibilidad del contenido y fortalece la resistencia frente a intentos de censura o interrupciones del servicio.

En este contexto, IPFS constituye una base tecnológica especialmente adecuada para el desarrollo de aplicaciones descentralizadas. A continuación, se presentan algunas de las herramientas más relevantes que conforman su ecosistema.

libp2p Colección de protocolos y utilidades para facilitar la implementación de una red *peer-to-peer* [60]. Entre sus herramientas, se encuentran diferentes mecanismos de seguridad, de transporte,

y para descubrimiento de pares. Se creó con IPFS en mente, pero luego se expandió a un conjunto de protocolos independiente, el cual es utilizado por Ethereum actualmente. Los protocolos de interés para este proyecto son:

Protocolos de transporte Son los encargados de la comunicación entre nodos, de manera similar a la capa de transporte presente en toda red convencional. Se basan en tipos de transporte ya existentes, adaptados al uso *peer-to-peer*. Los protocolos principales son TCP, WebSockets y WebRTCDirect.

Protocolos de descubrimiento de peers Para encontrar un contenido en IPFS, no es necesario conocer de antemano la dirección del nodo que lo almacena. Al contrario, alcanza con conocer el hash del contenido, y la red se encarga de localizar uno o más nodos que lo tengan disponible en ese momento. El principal protocolo utilizado para esto se denomina Distributed Hash Table (DHT) [29] [70]. Se trata de un registro clave-valor distribuido entre todos los nodos que soportan este protocolo, el cual contiene la información necesaria para encontrar el contenido deseado. Cada nodo mantiene una porción de esta tabla y puede realizar consultas a otros nodos para descubrir qué peer posee el contenido buscado.

Kubo La implementación principal de IPFS es Kubo, una solución hecha en Go [58]. Tiene su propio comando en la terminal, llamado `ipfs`, y también es utilizado en los demás *front-ends* de IPFS como *IPFS Desktop*, la aplicación de escritorio de IPFS. Es la más madura y desarrollada de las dos implementaciones de IPFS, y cuenta con más funcionalidades.

IPFS Cluster Es una herramienta diseñada para coordinar múltiples nodos de IPFS con el fin de facilitar la replicación y sincronización de contenidos en la red [55]. Su función principal es garantizar la alta disponibilidad y persistencia de los datos distribuidos, permitiendo que varios nodos colaboren para mantener y distribuir los archivos de manera eficiente y resiliente.

Clústeres colaborativos Se trata de configuraciones de clústeres que permiten la participación de usuarios mediante sus propios nodos para incrementar la disponibilidad y redundancia del contenido replicado [56]. Esta característica resulta particularmente adecuada para aplicaciones comunitarias, dado que posibilita la colaboración abierta sin comprometer la seguridad ni el control sobre los datos almacenados.

OrbitDB Es una base de datos descentralizada y *peer-to-peer* construida sobre **IPFS** para el almacenamiento distribuido de datos, y que utiliza **libp2p** para la sincronización y comunicación entre nodos [68]. Su modelo de consistencia es eventual, lo que implica que los cambios realizados en una instancia de la base de datos no se reflejan instantáneamente en todos los nodos, sino que se propagan progresivamente a lo largo del tiempo. OrbitDB ha sido desarrollada principalmente dentro del ecosistema JavaScript, lo cual facilita su integración en aplicaciones web y permite la ejecución de nodos directamente en el navegador.

Helia Helia es una implementación de IPFS en JavaScript que puede ejecutarse tanto en navegadores como en entornos de servidor [40]. A diferencia de Kubo, Helia ofrece un enfoque más abierto y flexible para la configuración del nodo *libp2p* que utiliza internamente. Esta flexibilidad permite adaptar el nodo a diversos escenarios y necesidades específicas, aunque también implica una mayor complejidad para garantizar un funcionamiento correcto y estable.

7.3.2. Blockchain

Es una tecnología basada en una cadena de bloques de operaciones descentralizada. Esta tecnología genera una base de datos compartida a la que tienen acceso sus participantes, los cuáles

pueden rastrear cada transacción que hayan realizado. En la actualidad, existen diversas implementaciones de *blockchains* que permiten la creación de aplicaciones descentralizadas.

Ethereum Es una de las *blockchains* públicas más utilizadas hoy en día [92]. Está compuesta de nodos distribuidos de manera descentralizada que comparten poder de cómputo sobre los cuales se desarrollan aplicaciones descentralizadas. Cuenta con una criptomoneda llamada ETH que funciona a modo de incentivo, es decir que los nodos reciben ganancias por formar parte de la red. Esto conlleva a que los usuarios de la red necesiten pagar para utilizarla a través de transacciones. Las aplicaciones en Ethereum funcionan mediante *smart contracts* inmutables que son desarrollados en el lenguaje Solidity.

Swarm Es un almacenamiento descentralizado que corre sobre una *sidechain* de Ethereum [86]. Surgió como uno de los tres pilares de Ethereum para una web descentralizada [69]. Funciona por *content addressing* como IPFS e incluye un modelo de incentivos utilizando su propia criptomoneda llamada BZZ.

7.3.3. Alternativas

Hyphanet Es una plataforma *peer-to-peer* para publicar y comunicar, resistente a la censura y respetuosa de la privacidad [20] [45]. Originalmente conocido como Freenet y creado como un trabajo profesional de fin de carrera por Ian Clarke, Hyphanet es una plataforma de software libre que permite compartir archivos, navegar y publicar sitios de forma anónima. Es descentralizado para hacerlo menos vulnerable a ataques, y de ser usado sólo con personas de confianza lo hace difícil de detectar para agentes externos.

Freenet Es una red *peer-to-peer* para servicios descentralizados, sin censura, en donde los usuarios tengan el control del contenido [37]. Es una plataforma que transforma las computadoras de sus usuarios en plataformas distribuidas y resilientes en la que se pueden construir aplicaciones descentralizadas. Cada *peer* contribuye a un colectivo tolerante a fallas, asegurando que los servicios robustos que estén siempre disponibles.

Creado por Ian Clarke, el mismo creador de *Hyphanet* [45], es una plataforma nueva que busca ser una computadora descentralizada en reemplazo de servidores centralizados. Está hecha en Rust y utiliza WebAssembly para ejecutar las aplicaciones.

7.4. Soluciones existentes

7.4.1. IPFS Deploy Action

Para desplegar un sitio web en IPFS, se puede utilizar un GitHub Action [27] desarrollado por IPFS y lanzado en Febrero de 2025. Dicha herramienta es un *script* que se ejecuta con cada *commit* en un repositorio de GitHub, y permite compilar el sitio web, y luego alojarlo en IPFS utilizando un servicio de *pinning*, Filecoin, o bien un clúster de IPFS. Estas opciones se verán en detalle cuando se abarque la arquitectura de despliegue implementada.

Sin embargo, cuenta con una serie de desventajas al utilizar un *cluster*. Por un lado, utilizando la opción de *cluster* sólo puede instruir a un único nodo para que luego este sincronice el contenido al resto de los nodos. Un conjunto de nodos que colaboran con el despliegue y mantenimiento de un sitio web sólo podrán actualizarse si el nodo central está activo, lo que resulta en un único punto de falla. Por otro lado, utilizar un GitHub Action nos ata necesariamente a GitHub, lo cuál supone otra limitación para quienes deseen optar por otra alternativa para su repositorio Git.

7.4.2. Distributed Wikipedia Mirror

Una iniciativa destacada en este ámbito es el proyecto Distributed Wikipedia Mirror [30], que replica una versión estática de Wikipedia dentro de la red IPFS. Su objetivo principal es garantizar el acceso inalterable y resistente a la censura del contenido enciclopédico. Sin embargo, esta solución funciona únicamente en modo *read-only* y se basa en *snapshots* generados manualmente, lo que limita su dinamismo y capacidad de actualización en tiempo real. A pesar de estas limitaciones, demuestra la viabilidad de alojar grandes volúmenes de información en IPFS y representa un caso de uso muy cercano al repositorio de conocimiento que buscamos construir.

8. Problema detectado y/o faltante

Las soluciones de infraestructura actuales, como AWS, Azure o Google Cloud, presentan barreras significativas para su adopción por parte de proyectos con recursos limitados, como iniciativas independientes, educativas o comunitarias. Estas barreras están relacionadas principalmente con los costos operativos, la dependencia de infraestructura centralizada y la vulnerabilidad frente a fallas o interrupciones del servicio.

8.1. Costo y sostenibilidad

La mayoría de las aplicaciones con requerimientos de alta disponibilidad dependen de servicios centralizados en la nube, los cuales implican costos mensuales elevados incluso en etapas tempranas del desarrollo. Este modelo económico desalienta la creación y mantenimiento de servicios comunitarios o de bajo presupuesto, restringiendo su escalabilidad o continuidad en el tiempo.

A su vez, la dependencia de servidores centralizados genera un punto único de mantenimiento y financiamiento que puede convertirse en un cuello de botella. En escenarios donde no existe un respaldo institucional o comercial fuerte, la sostenibilidad del servicio queda comprometida.

8.2. Interrupciones e infraestructura crítica

La arquitectura tradicional basada en cliente-servidor implica una fuerte dependencia de la disponibilidad continua de uno o varios nodos centrales. Estas arquitecturas son susceptibles a interrupciones por mantenimiento, errores de configuración, fallas de *hardware* o problemas de conectividad. En muchos casos, un único incidente puede volver una aplicación completamente inaccesible para todos sus usuarios.

Esto evidencia la necesidad de diseñar infraestructuras más resistentes a fallas, donde la continuidad operativa no dependa de un único punto de control.

8.3. Centralización y control de acceso

La centralización también facilita el control externo sobre los servicios. Aplicaciones y plataformas pueden ser bloqueadas o restringidas en determinados contextos geográficos o políticos simplemente mediante la identificación y bloqueo de sus puntos de acceso. Esto representa una amenaza para la disponibilidad global y el acceso libre a herramientas comunitarias.

Un ejemplo representativo es el de Wikipedia, cuyo acceso ha sido bloqueado —de forma total o parcial— en diversos países, debido a restricciones impuestas sobre determinados contenidos [19].

8.4. Accesibilidad tecnológica

Finalmente, muchas de las tecnologías necesarias para implementar infraestructuras distribuidas aún requieren conocimientos técnicos avanzados para su configuración, despliegue y operación.

Esta complejidad técnica representa una barrera de entrada tanto para usuarios como para desarrolladores que podrían beneficiarse de este tipo de arquitectura, pero que no cuentan con los recursos o conocimientos necesarios para adoptarla.

9. Solución implementada

Se implementaron tres casos de uso sobre los distintos ecosistemas a analizar. A continuación, se detallan las decisiones técnicas adoptadas en cada implementación, haciendo uso de las diversas herramientas disponibles en cada ecosistema para cumplir con los requisitos funcionales presentados. Para luego presentar un análisis cualitativo y cuantitativo de las ventajas y desventajas de cada caso, según el ecosistema.

9.1. Casos de uso

9.1.1. Sitio web estático

Este caso de uso consiste en desplegar un sitio o aplicación web, para poder visualizarlo mediante un navegador. Es uno de los casos más simples y sirve como introducción para familiarizarse con cada ecosistema.

Requisitos funcionales

- **Interfaz de usuario**: sitio web estático que contenga la interfaz de usuario principal a los casos de uso y sus ecosistemas de despliegue.

Aplicación Este caso de uso se implementó como **Astrawiki-web** [9]. En las subsiguientes secciones, las menciones a Astrawiki-web referirán a este caso de uso.

9.1.2. Repositorio de conocimiento

Este caso representa un repositorio con diferentes artículos, similar a *Wikipedia*. Es un servicio comunitario en donde se puede agregar información de distinta índole. Con este caso se analizó la capacidad de creación, modificación y recuperación de contenido por parte de los usuarios.

Requisitos funcionales

- **Edición:** los artículos dentro del repositorio deben poder editarse por cualquier persona que ingrese al sitio, y este cambio debe verse reflejado eventualmente en las demás personas que accedan a ese artículo.
- **Historial de versiones:** cada artículo debe tener una lista de versiones anteriores, junto con hipervínculos con los cuáles acceder a ellas.
- **Búsqueda:** una persona debe poder realizar una búsqueda global que incluya todos los artículos.

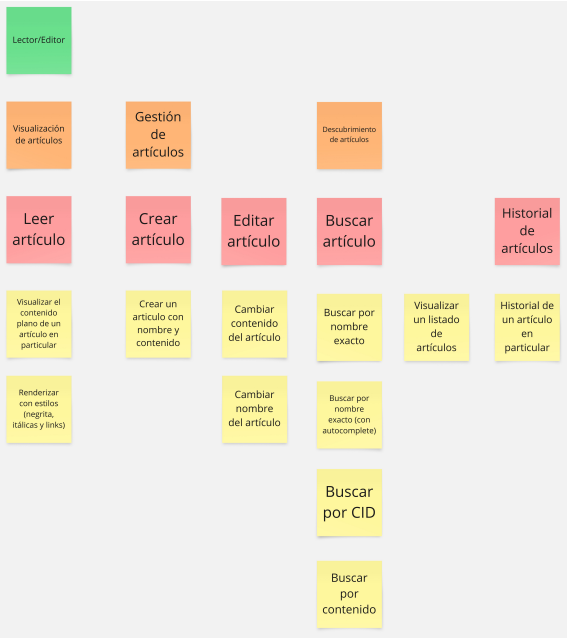

Figura 1: *User Story Map* del repositorio de conocimiento

Package Este caso de uso se implementó en los *packages* **Astrawiki** [8] (implementación sobre el ecosistema IPFS) y **Astrawiki-eth** [7] (implementación sobre el ecosistema Blockchain). En las subsiguientes secciones, las menciones a Astrawiki referirán a este caso de uso.

9.1.3. Mensajero en tiempo real

Este caso se enfoca en la capacidad de la infraestructura de enfrentarse a situaciones de *tiempo real* como puede ser un chat de texto o de audio. En particular, nos centramos en el caso de chats de texto para un grupo de usuarios en donde los mensajes sean públicos.

Requisitos funcionales

- **Usuarios:** se deben contar con usuarios que puedan iniciar sesión con una clave.
- **Grupos públicos:** grupos de chat de texto, donde cualquier usuario puede ingresar y ver los mensajes del resto, así como también participar enviando sus propios mensajes.
- **Respuestas:** un usuario debe poder responder mensajes anteriores dentro de un mismo chat.

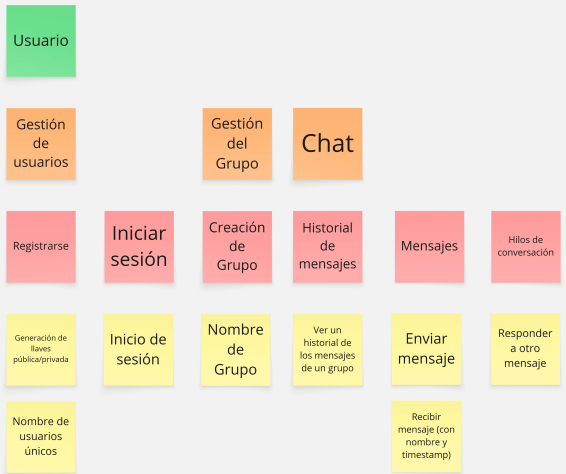

Figura 2: *User Story Map* del mensajero en tiempo real

Package Este caso de uso se implementó en los *packages* **Astrachat** [4] (implementación sobre el ecosistema IPFS) y **Astrachat-eth** [3] (implementación sobre el ecosistema Blockchain). En las subsiguientes secciones, las menciones a Astrachat referirán a este caso de uso.

9.2. IPFS

Durante el desarrollo de los casos de uso sobre el ecosistema IPFS, se fue consolidando una comprensión más profunda de los componentes necesarios para construir aplicaciones distribuidas y descentralizadas. Esto permitió identificar y diseñar una serie de abstracciones que encapsulan la infraestructura general requerida, facilitando la implementación de los casos de uso sobre una base común.

Se distinguen dos infraestructuras principales en funcionamiento: la infraestructura de despliegue y la infraestructura de aplicación.

La **infraestructura de despliegue** se encarga del alojamiento de las aplicaciones web. Es responsable de distribuir el *front-end* de las aplicaciones, así como también el código necesario para su ejecución en caso de que incluyan componentes dinámicos. Toda aplicación que deba ser accesible desde la web hace uso de esta infraestructura, como ocurre con el sitio web estático.

Por otro lado, la **infraestructura de aplicación** gestiona la lógica de la aplicación, incluyendo el almacenamiento de datos y la conexión entre pares. Esta infraestructura es fundamental para aplicaciones que requieren mantener un estado compartido entre usuarios y permitir su modificación, como en el caso del repositorio de conocimiento y el mensajero en tiempo real.

La identificación de estas abstracciones permitió encapsular gran parte de la complejidad asociada al ecosistema subyacente, lo que simplificó el desarrollo tanto de los casos de uso implementados como de futuros desarrollos. Gracias a esta separación, es posible concentrarse exclusivamente en los requisitos funcionales de cada nuevo caso de uso, sin necesidad de interactuar directamente con los detalles técnicos de IPFS y sus herramientas subyacentes.

A continuación, se detalla la composición y el funcionamiento de ambas infraestructuras.

9.2.1. Infraestructura de despliegue

El objetivo de la infraestructura de despliegue fue alojar los archivos estáticos de un sitio web en IPFS y garantizar su acceso mediante un navegador, asegurando su correcta interpretación. Asimismo, se priorizó una filosofía basada en aplicaciones estrictamente comunitarias para la selección de las tecnologías y herramientas del ecosistema empleadas.

Para empezar, se debe explicar brevemente el concepto de *pinning*. Al subir un archivo a IPFS, este se procesa mediante una función de *hash*, y así se obtiene un *Content Identifier* (CID) único. Desde ese momento, cualquier nodo que desee obtener el archivo puede encontrarlo utilizando dicho CID. Sin embargo, no se asegura la persistencia del archivo, y dejará de ser accesible luego de un tiempo. Por esta razón existe el concepto de *pinning* [73], que consiste en fijar un archivo o directorio para tratar dicha información como esencial y, por lo tanto, evitar que el nodo lo descarte eventualmente. Este concepto es central para la solución implementada.

A continuación se describen brevemente las herramientas utilizadas.

Clústeres colaborativos en IPFS Un *clúster* es un grupo de nodos de IPFS que actúan en conjunto para fijar contenidos. Funcionan sincronizando su *pin set*, o sea, su lista de archivos y directorios fijados en un momento dado. Un clúster *colaborativo* sigue esta premisa, pero permite que los usuarios colaboren con su nodo local sin posibilidad de modificar los archivos, lo cuál se delega a nodos especiales con la capacidad de orquestar el clúster en conjunto [56]. Así, se logra que la misma comunidad mantenga operativo el mecanismo de despliegue de la aplicación, en línea con la filosofía de aplicaciones comunitarias.

IPNS El CID cambiará si el contenido del sitio web o aplicación web cambia. Este problema puede ser resuelto con la ayuda de *punteros mutables*, objetos de IPFS que apuntan a un CID específico y pueden ser modificados. Esto permite compartir la dirección del puntero una única vez y actualizar el CID al que apunta cada vez que se haga un cambio. **IPNS** (InterPlanetary Name System) es un sistema que permite crear punteros mutables y obtener una dirección conocida como *nombre de IPNS* [54]. Estos nombres de IPNS pueden considerarse como enlaces que pueden actualizarse, conservando al mismo tiempo la verificabilidad del *content addressing*. El titular de la clave privada puede firmar y publicar nuevos registros en cualquier momento.

Una vez que se aloja un contenido en IPFS y se apunta a él mediante un *nombre* de IPNS, el mayor problema pasa a ser la manera de acceder a IPNS en sí. El hecho de que los nombres sean *hashes* alfanuméricos, y no nombres legibles o memorables para humanos, dificulta el acceso de los usuarios a un sitio web memorable y fácil de compartir. A continuación se analizará la alternativa elegida para solucionar este problema.

```
/ipns/k51qzi5uqu5dhkdbjdsauuyk5iyq82uzpjb0is3x6oy9dcmmr8dbcezv7v9fya
```

Figura 3: Ejemplo de la dirección de un nombre de IPNS

Dado que el valor al que apunta el nombre de IPNS cambia con cada modificación del proyecto, se vuelve necesario seleccionar un grupo de nodos que se les confíe con tal fin. De lo contrario un posible atacante podría modificar el registro, invalidándolo o redirigiéndolo. Por la misma razón, no cualquier nodo dentro del clúster debe ser capaz de cambiar el *pin set*, o lista de CIDs fijados por el clúster.

IPFS Cluster tiene en cuenta esto, y hace la distinción entre un nodo *trusted* y un nodo *follower* para su implementación de clústeres *colaborativos* [22]. Para esta herramienta, se utiliza las denominaciones de nodo confiable y nodo colaborador, respectivamente.

Nodo confiable Este nodo tiene la capacidad de modificar el nombre IPNS, como también actualizar la configuración del mismo, y el *pin set*. Son una parte esencial del clúster, ya que ellos no es posible modificar el contenido. Esto no implica centralización, ya que la comunidad puede formar sus propios grupos de nodos confiables y actualizar el contenido por su cuenta, similar a realizar un *fork* en un proyecto de Github.

Nodo colaborador Únicamente se encarga de fijar los archivos establecidos por los nodos confiables, y actualizar su *pin set* cuando se lo indique. Al igual que los nodos confiables, debe fijar

la totalidad de los archivos. Su finalidad es aumentar la disponibilidad del contenido y evitar que la información se pierda.

En un escenario ideal, existen varios nodos confiables disponibles simultáneamente. Esto previene un posible *single point of failure* y asegura la continuidad y validez del clúster. Además, aunque los nodos confiables no estén en línea, los nodos colaboradores mantienen la última actualización del clúster, garantizando así la disponibilidad y persistencia del contenido.

ENS Ethereum Name Service (ENS) es un protocolo de nombres descentralizado que se basa en la *blockchain Ethereum* [33]. Funciona de manera similar a DNS, en el sentido de que traduce identificadores como nombres y *hashes* a formatos legibles para humanos. Al operar sobre la *blockchain* de Ethereum, es seguro, descentralizado y transparente. Está diseñado específicamente para traducir identificadores como direcciones de billeteras, *hashes* y metadatos, entre otros, incluyendo direcciones de IPFS. Es posible configurar un registro ENS que resuelva automáticamente una dirección IPNS. Esto permite usar nombres legibles, más fáciles de compartir y acceder, y soluciona el principal problema de IPNS hasta este punto. Además, cuando se quiera actualizar el contenido, no será necesario modificar el registro ENS en sí, ya que el registro siempre apuntará al mismo nombre de IPNS.

Gateways Los navegadores modernos no siempre soportan IPFS o ENS de forma nativa. Algunos, como Opera [14] o Brave [17], permiten acceder directamente a direcciones IPFS o dominios `.eth`. Sin embargo, la mayoría no ofrece esta funcionalidad, lo que limita el acceso a contenido descentralizado.

Para superar esta limitación, se emplean *gateways* de Ethereum e IPFS. Un *gateway* de Ethereum traduce un dominio `.eth` a una dirección IPNS, mientras que un *gateway* de IPFS recupera el contenido asociado desde la red distribuida a través de solicitudes HTTP. Algunas *gateways* tienen la funcionalidad de mostrar una página web de manera correcta cuando un directorio tiene la estructura indicada. Esto resulta particularmente útil para poder mostrar una página moderna de la misma manera que se haría utilizando un servidor HTTP.

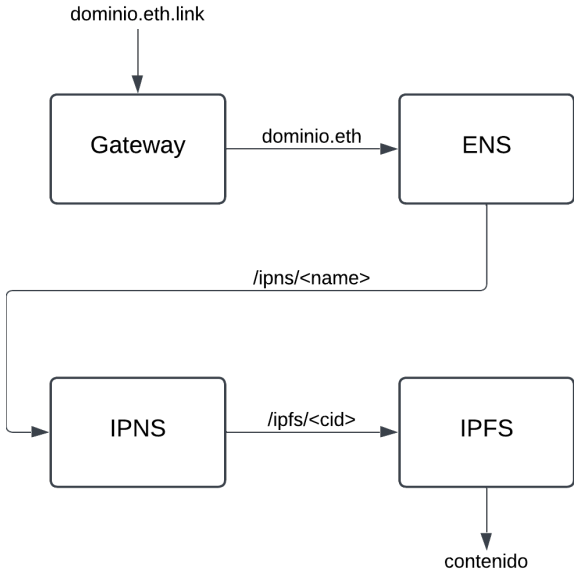

Figura 4: Mapa de la traducción de un dominio al contenido de IPFS

Para nuestro caso, se utilizó el servicio de Limo [61], que soporta direcciones de ENS, y recupera archivos de IPFS, actuando como *gateway* para ambos servicios. Para utilizarlo, simplemente se requiere agregar un sufijo `.link` al nombre de ENS. Por ejemplo, `dominio.eth` puede accederse a través de `dominio.eth.link`. A su vez, resuelve `dominio.eth` a una dirección de IPFS, y por lo

tanto obtiene los archivos de esa dirección automáticamente.

Si bien las *gateways* mejoran la accesibilidad desde navegadores tradicionales, también introducen un grado de centralización. Sin embargo, existen múltiples servicios de *gateways*, tanto de IPFS como de Ethereum, que se pueden utilizar si una de ellas falla.

Despliegue continuo Para lograr un despliegue continuo que integre automáticamente un repositorio Git, se diseñó un *script* que se ejecuta de forma autónoma en cada nodo confiable del clúster. En esta arquitectura mayormente horizontal, no existe un servidor central que orqueste las actualizaciones. Esto elimina un punto único de falla y favorece la resiliencia del sistema. Por lo tanto, cualquier nodo confiable debe poder actualizar su contenido e instruir a los nodos colaboradores para que actualicen su contenido de manera equivalente. Además, el sistema debe funcionar correctamente incluso si los nodos no reciben las actualizaciones simultáneamente, evitando así *race conditions*, es decir, situaciones donde varios procesos intentan modificar un recurso al mismo tiempo.

Como se mencionó anteriormente, debido a la naturaleza del direccionamiento basado en contenido de IPFS, si dos nodos suben el mismo contenido, obtendrán el mismo CID. Esta propiedad permite que cualquier nodo confiable actualice el contenido y el nombre de IPNS de forma independiente a los demás nodos confiables. Cuando un nodo detecta un cambio nuevo, obtiene el código estático actualizado y, acto seguido, indica a los nodos colaboradores que fijen el CID correspondiente. Si el nodo es el primero en detectar dicha actualización, también instruye al resto del clúster para que dejen de fijar el CID antiguo.

Compilado Las herramientas de compilación no siempre generan archivos idénticos en compilaciones sucesivas a partir del mismo código fuente, debido a factores como la inclusión de marcas temporales o *hashes* internos. Por ejemplo, `Next.js` produce archivos estáticos diferentes incluso cuando el código fuente no cambia, lo cual representa un problema para el enfoque propuesto, ya que dos nodos que compilen localmente el mismo código podrían obtener CIDs distintos y, por ende, no sincronizar correctamente el contenido.

Para mitigar esta situación, se implementó un *hook* automatizado, integrado en el flujo de trabajo del repositorio, que compila el código una única vez por cada *commit* en la rama principal. De este modo, los nodos confiables pueden detectar los cambios en la rama que contiene los archivos compilados, realizar un *pull* de esos archivos ya generados y garantizar que todos obtengan el mismo CID.

Configuración Para que un usuario pueda conectarse y contribuir como nodo colaborador a un clúster, necesita acceder a una dirección de IPFS desde la cual obtener el archivo `service.json` [23]. Este archivo de configuración contiene la información necesaria para integrarse correctamente al clúster, incluyendo las *multiaddresses* [65] de los nodos confiables. Dado que estos nodos pueden agregarse o eliminarse, el archivo está sujeto a modificaciones, por lo que debe ser tratado como parte integral del proceso de despliegue. Cada vez que se actualiza el archivo —por ejemplo, tras un cambio en el repositorio Git donde se aloja—, se debe fijar nuevamente en el clúster y actualizar el nombre de IPNS asociado. De esta forma, los usuarios siempre podrán acceder a la versión actualizada del archivo de configuración mediante un enlace distribuible y verificable.

```
/ip4/123.123.123.123/udp/9096/quic/p2p/12D3KooWLw...yPcuZJR
```

Figura 5: Ejemplo de una *multiadress* posible que utiliza el protocolo QUIC.

Limitaciones Este enfoque, si bien permite un despliegue descentralizado y gestionado por la comunidad, presenta algunas limitaciones técnicas y operativas:

Actualización completa del contenido Cada vez que se realice un cambio en el directorio de la aplicación, se deberá fijar una nueva versión completa del contenido. Esto obliga a los nodos colaboradores a descargar nuevamente todo el directorio, lo que puede implicar un costo elevado en términos de ancho de banda y tiempo, especialmente para archivos de gran tamaño.

Problemas de cacheo en IPNS El parámetro TTL o *time-to-live* de IPNS determina cuánto tiempo permanece un valor en la caché de un nodo antes de consultarse nuevamente en la *Distributed Hash Table* (DHT). Si se configura un TTL alto, los cambios recientes pueden demorar en propagarse. En cambio, un TTL bajo reduce la latencia de actualización pero incrementa el número de consultas a la red, lo cual puede afectar el rendimiento. No obstante, cada nodo mantiene siempre la última versión del nombre que haya resuelto correctamente.

Distribución de claves privadas Como las actualizaciones de IPNS requieren estar firmadas con una clave privada, todos los nodos confiables deben compartir la misma clave para poder emitir registros válidos. Esto elimina el punto único de falla, pero al mismo tiempo incrementa el riesgo de exposición si la clave es comprometida.

Requerimiento de apertura de puertos La operación de IPFS Cluster exige la apertura del puerto 9096 para la comunicación entre nodos. Este requerimiento puede representar una barrera para ciertos usuarios, especialmente aquellos con configuraciones de red restrictivas o sin conocimientos técnicos avanzados.

Implementación del nodo confiable En base a este análisis, podemos concluir que la mejor forma de desplegar una página web estática en IPFS es a través del uso de un clúster colaborativo compuesto por nodos confiables que se integren con el proyecto de Git dado, así como una dirección IPNS a la cuál actualizar cada vez que hay un cambio, y un registro ENS para traducir la dirección IPNS a un nombre legible.

Arquitectura general La herramienta está compuesta por tres contenedores:

- **Kubo:** el nodo de IPFS encargado de conectarse a la red de IPFS para publicar y obtener el contenido necesario.
- **IPFS Cluster:** gestiona el contenido fijado y coordina con otros nodos del clúster.
- **Watcher:** observa los repositorios de Git del proyecto y del archivo `service.json`, y orquesta acciones en los otros dos contenedores.

La comunicación entre contenedores se realiza a través de sus respectivas APIs HTTP [44] [43]. El contenedor watcher está basado en Alpine Linux y utiliza scripts POSIX-compliant para máxima portabilidad.

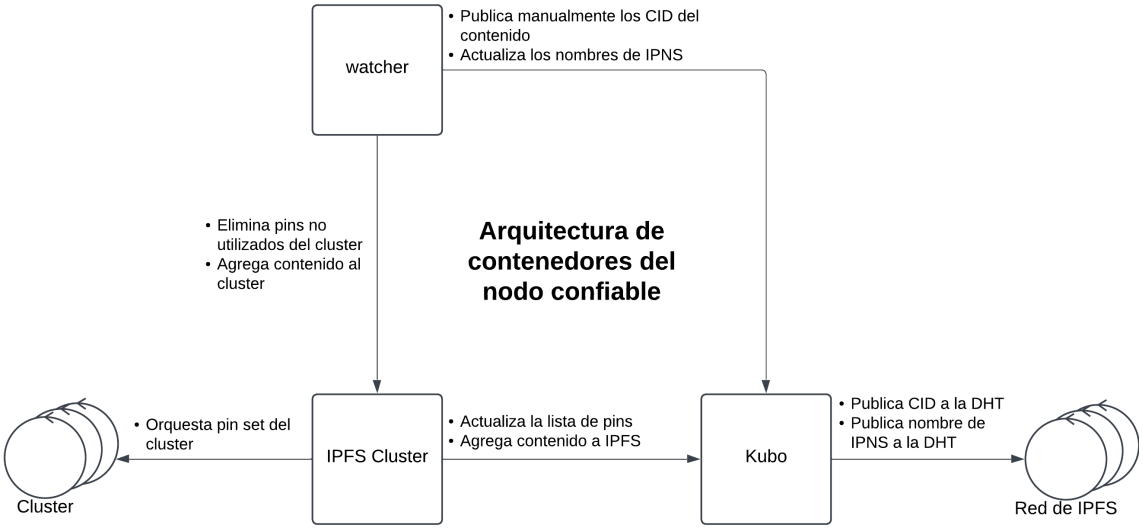

Figura 6: Mapa de interacciones entre los contenedores del nodo confiable

Pipeline de despliegue El contenedor *watcher* compara periódicamente el último *commit* de una rama remota con una copia local clonada al iniciar. Al detectar un cambio en el contenido o en el archivo `service.json`, ejecuta el siguiente flujo:

1. Sube el contenido y el archivo `service.json` al clúster, obteniendo sus respectivos CIDs.
2. Publica manualmente ambos CIDs en la red IPFS usando Kubo.
3. Verifica que todos los nodos del clúster hayan fijado los nuevos CIDs.
4. Actualiza los registros IPNS correspondientes.
5. Elimina los *pins* antiguos del clúster para liberar recursos.

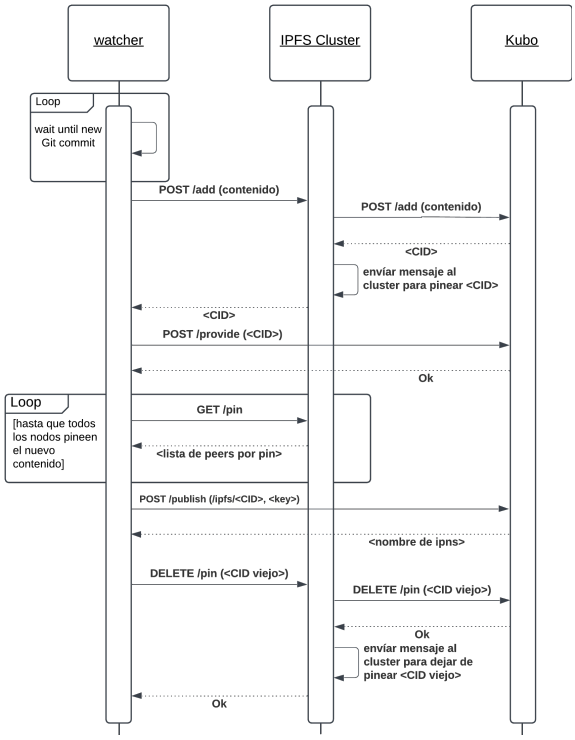

Figura 7: Diagrama de secuencia para el caso en que *watcher* detecta un cambio. Notar que para mayor claridad se omite los pasos para desplegar el nombre de IPNS del `service.json`, al ser exactamente los mismos que en el caso de un contenido.

Gestión de claves de IPNS Cada nodo confiable requiere una identidad persistente, definida por un identificador de nodo o `PeerID` y su clave privada, que debe conservarse entre ejecuciones. Esta identidad es la que figura en el archivo `service.json`, el cual los nodos colaboradores utilizan para unirse al clúster.

Además, todos los nodos confiables deben compartir las mismas claves privadas de IPNS para poder actualizar los registros. Se utilizan dos claves distintas: una para el contenido estático y otra para `service.json`. Un *script* interactivo facilita la generación inicial de estas configuraciones, incluyendo identidad, claves IPNS, direcciones de repositorios y demás parámetros necesarios.

Disponibilidad Dos factores pueden afectar la disponibilidad del contenido:

- **Propagación del IPNS:** Un nuevo valor puede no estar aún distribuido en toda la DHT. Por ello, se garantiza primero la publicación exitosa del nuevo valor antes de eliminar el contenido anterior del clúster.
- **Indexación del nuevo CID:** Publicar el contenido manualmente mediante Kubo antes de actualizar el IPNS asegura que esté accesible de forma inmediata desde *gateways* o nodos externos.

Este enfoque añade una latencia al proceso de actualización, pero asegura la disponibilidad continua de al menos una versión válida del contenido.

Resultado La solución permite instanciar un nodo confiable mediante un único comando (`make up`), el cual se encarga automáticamente del monitoreo, despliegue y publicación del contenido en IPFS. Aunque está optimizada para aplicaciones web, la herramienta es agnóstica respecto al tipo de contenido, siendo apta también para documentación, recursos estáticos o repositorios.

Combinada con un registro ENS y un gateway compatible, esta solución ofrece una experiencia de usuario comparable a la de un servidor HTTP moderno, pero con las ventajas inherentes de una infraestructura comunitaria, descentralizada y de bajo costo.

9.2.2. Infraestructura de aplicación

Las aplicaciones dinámicas, aquellas que requieren mantener un estado mutable, como un repositorio de conocimiento o un mensajero en tiempo real, presentan desafíos particulares en el ecosistema de IPFS. Si bien IPFS permite mantener un estado compartido mediante la publicación continua de nuevos CIDs y el uso de mecanismos como IPNS —el cual introdujimos y utilizamos en la infraestructura de despliegue—, **no está diseñado específicamente para la actualización eficiente de datos dinámicos en tiempo real**.

Esto introduce ciertas limitaciones a la hora de construir aplicaciones interactivas, donde múltiples usuarios necesitan leer y escribir datos de forma concurrente. En particular, la falta de mecanismos nativos para gestionar versiones, sincronizar cambios o evitar conflictos entre ediciones simultáneas hace que desarrollar aplicaciones con estado mutable directamente sobre IPFS resulte poco conveniente.

Para superar estas limitaciones, desarrollamos **AstraDB**, una infraestructura pensada para facilitar la creación y gestión de aplicaciones dinámicas y colaborativas dentro del ecosistema de IPFS.

A continuación, describiremos los componentes y la arquitectura de **AstraDB**, cómo permite representar entidades como artículos o conversaciones, y cómo se adapta a las necesidades de aplicaciones comunitarias y descentralizadas, tales como wikis colaborativas y sistemas de mensajería en tiempo real.

AstraDB AstraDB es una infraestructura diseñada para facilitar el desarrollo de aplicaciones comunitarias, distribuidas y descentralizadas que requieren mantener un estado mutable. Construida como una capa superior a **OrbitDB** [68], abstrae muchos de los aspectos de bajo nivel necesarios para operar bases de datos en redes *peer-to-peer*.

Su diseño prioriza la colaboración entre múltiples usuarios, permitiendo que cualquier persona pueda contribuir activamente al sostenimiento de la aplicación sin depender de servidores centrales ni infraestructura dedicada. Para ello, automatiza tareas fundamentales como la sincronización, el descubrimiento de nodos y la replicación de datos entre colaboradores.

AstraDB expone una interfaz simple inspirada en el modelo de pares *clave-valor*, donde cada entidad —como una conversación o un artículo— se asocia a una clave única. Las operaciones principales permiten agregar nuevos valores bajo una clave, consultar los valores existentes y suscribirse a actualizaciones en tiempo real, facilitando así el desarrollo de aplicaciones colaborativas sin exponer al desarrollador a la complejidad del entorno distribuido.

Representación de los datos AstraDB utiliza OrbitDB para la representación de sus datos. OrbitDB es una base de datos distribuida *peer-to-peer* que utiliza IPFS para el almacenamiento y **libp2p** [60] para la sincronización entre nodos. Es una base de datos *eventualmente consistente*, diseñada específicamente para operar en redes descentralizadas sin depender de servidores centrales, lo que la hace especialmente adecuada para aplicaciones distribuidas.

OrbitDB ofrece distintos tipos de bases de datos, adaptadas a diversos modelos y casos de uso, como bases de documentos o estructuras de eventos secuenciales.

Con el objetivo de garantizar una verdadera descentralización y evitar que exista una entidad con permisos privilegiados, AstraDB utiliza bases de datos del tipo de eventos, las cuales son *append-only*. Este tipo de base de datos mantiene un historial inmutable de entradas, donde los nodos pueden únicamente agregar nuevos registros sin posibilidad de modificar o eliminar los existentes. Esto asegura la integridad y preservación de la información, permitiendo que cualquier usuario pueda contribuir sin comprometer la consistencia del sistema.

Esta decisión de trabajar con estructuras *append-only* introduce, sin embargo, un nuevo desafío: ¿cómo representar información actualizable si no se puede sobrescribir el valor de una clave?

La solución adoptada por AstraDB consiste en asignar a cada clave su propia base de datos de eventos. De esta forma, por ejemplo, un artículo puede representarse como una secuencia de actualizaciones almacenadas en orden cronológico. Cada entrada en la base corresponde a un cambio realizado sobre esa entidad, y su historia completa permite reconstruir su versión más reciente o auditar su evolución a lo largo del tiempo.

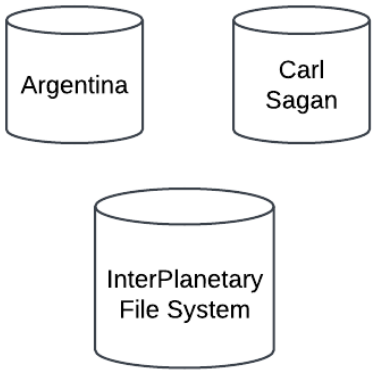

Figura 8: Cada entidad se representa con su propia base de datos de eventos.

Este enfoque también permite una distribución eficiente: dado que en OrbitDB cada base de datos debe ser replicada localmente para poder ser leída, AstraDB permite que cada nodo sincronice únicamente las claves (y, por ende, las bases) que le interesan. Esto reduce significativamente la carga de almacenamiento y comunicación.

Para registrar qué claves existen dentro de una instancia de AstraDB —por ejemplo, qué artículos están presentes en un repositorio de conocimiento— se utiliza una base de datos adicional, también del tipo de eventos, que funciona como índice global. Allí se agregan nuevas claves a medida que se crean, permitiendo a los nodos descubrir las entidades disponibles sin necesidad de conocer todo el sistema de antemano.

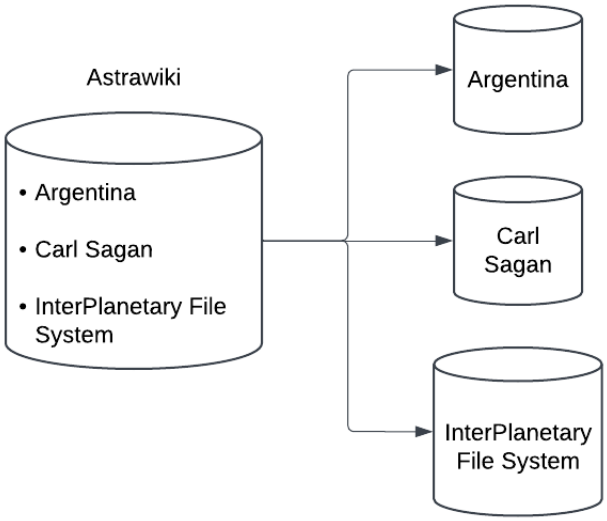

Figura 9: Base de datos de eventos que actúa como índice global de claves.

Identificación y acceso a los datos Cada base de datos en AstraDB se identifica a través de una dirección única generada por OrbitDB. Esta dirección está compuesta por tres elementos: el tipo de base, su controlador de acceso y un nombre identificador. Dado que AstraDB utiliza siempre el mismo tipo (eventos) y un controlador abierto —que permite a cualquier usuario agregar información—, la dirección final queda determinada únicamente por el nombre elegido.

Esta propiedad resulta fundamental para el funcionamiento del sistema. Permite que cualquier nodo, con solo conocer el nombre de una entidad, pueda acceder a la base de datos y sincronizarse con el resto de los nodos que ya la están replicando. Esto se debe a que crear una base de datos con el mismo nombre no genera una nueva, sino que devuelve la misma dirección y, por lo tanto, la misma base.

```
/orbitdb/zdpuAmrcSRUhkQcnRQ6p4bphs7DJWGBkqczSGFYynX6moTcDL
```

Figura 10: Ejemplo de dirección de una base de datos en OrbitDB.

AstraDB extiende este mecanismo de identificación para construir jerarquías lógicas de entidades. Por ejemplo, el nombre de una base puede componerse como `wiki:articulo`, lo que permite organizar de forma estructurada grandes cantidades de datos.

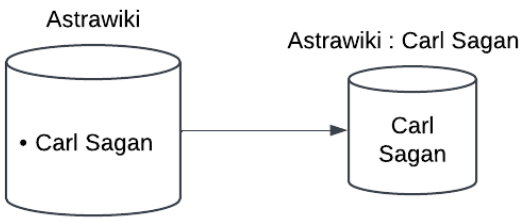

Figura 11: Estructura jerárquica de claves identificadoras.

Este enfoque también permite que múltiples instancias de AstraDB convivan de forma com-

pletamente independientes. La identidad de cada instancia —es decir, el conjunto de claves y sus respectivas bases de valores— queda determinada por el nombre identificador utilizado para crear su base de datos principal. De este modo, es trivial crear nuevas aplicaciones, wikis o sistemas colaborativos simplemente utilizando nombres distintos, sin generar colisiones ni interferencias con otras instancias.

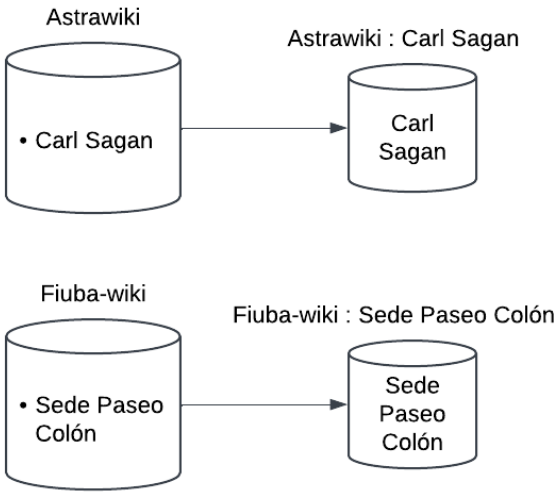

Figura 12: Representación de múltiples instancias independientes de AstraDB.

Sincronización y manejo de eventualidad En OrbitDB la sincronización entre nodos ocurre de forma eventual, lo que significa que, con el tiempo, todas las instancias de una misma base de datos tenderán a contener la misma información, pero no necesariamente al mismo tiempo.

OrbitDB emite eventos cuando una base de datos es actualizada, lo cual resulta crucial para AstraDB, especialmente al sincronizarse con otros nodos y recibir nuevas entradas. Sin embargo, debido a la naturaleza eventual de la consistencia que caracteriza a OrbitDB, y aunque se esté utilizando una base de datos de tipo evento secuencial, la entrada que desencadena el evento de actualización no siempre representa la última inserción cronológica. Es posible que, tras la sincronización, contenido aparezca antes o después de entradas más recientes en la estructura interna de la base de datos.

Para abordar esta situación, AstraDB implementa una capa de abstracción sobre OrbitDB que permite detectar todas las entradas nuevas, independientemente de su posición. Esta abstracción mantiene un registro de los identificadores *hash* de las entradas previamente vistas, y al recibir un evento de actualización, recorre toda la base de datos en busca de nuevas entradas aún no notificadas, emitiendo un evento por cada una de ellas.

Este mecanismo resulta especialmente importante en aplicaciones como el mensajero en tiempo real, donde es necesario garantizar que ningún mensaje pase desapercibido, incluso si fue insertado en una posición intermedia de la base de datos como resultado de la fusión de dos instancias previamente desincronizadas.

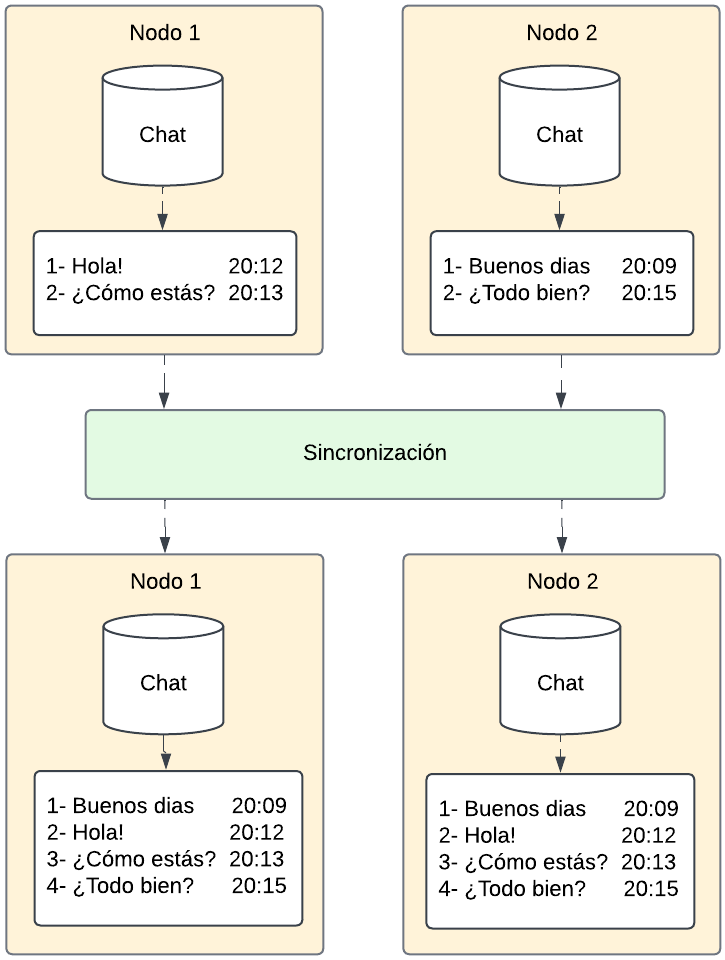

Figura 13: Ejemplo de sincronización entre nodos.

La Figura 13 ilustra este fenómeno: dos nodos mantienen versiones distintas de una misma base de datos denominada 'Chat'. Tras la sincronización, OrbitDB fusiona ambas versiones, generando una nueva instancia consistente. No obstante, desde la perspectiva del nodo 1, las nuevas entradas pueden no encontrarse exclusivamente al final de la base de datos, y la única notificación automática emitida por OrbitDB corresponde a la última entrada añadida. Por ello, es indispensable recorrer toda la base para garantizar la detección completa de las entradas incorporadas durante la sincronización.

Colaboradores En AstraDB, cada instancia existente de base de datos de OrbitDB —al abrirse— debe encontrar a otros nodos que ya la estén replicando para poder sincronizar su contenido. Si no hay ninguno disponible, la base aparece vacía, lo que implica una posible pérdida de información si no hay nodos que la conserven.

Para resolver este problema, AstraDB introduce el concepto de colaboradores: nodos que optan por conservar y replicar activamente todas las bases de datos existentes. En el caso del repositorio de conocimiento, por ejemplo, esto implica mantener localmente cada uno de los artículos publicados en el sistema.

La existencia de al menos un colaborador conectado garantiza que cualquier usuario pueda acceder al contenido completo de una base. De este modo, la red no depende de una infraestructura centralizada, sino de la voluntad de los usuarios de participar activamente en la preservación de la

aplicación.

A diferencia de otros enfoques distribuidos basados en incentivos económicos, AstraDB se apoya en un modelo comunitario: cualquier persona puede convertirse en colaborador, sin necesidad de permisos especiales ni recompensas. Su única función es brindar persistencia a las bases que almacena, ayudando a que el sistema permanezca accesible.

Conectividad Para que la sincronización entre distintas bases de datos de OrbitDB ocurra, primero deben estar ambos pares conectados. Dada la naturaleza de consistencia eventual con la que trabaja OrbitDB, este no se responsabiliza por el establecimiento ni el mantenimiento de las conexiones entre nodos, simplemente asume que, si dos nodos están conectados, eventualmente sincronizarán sus réplicas. Por eso, en su lugar, delega completamente esta tarea a la capa subyacente de red, que en este caso está conformada por **LibP2P** [60], a través de la implementación provista por **Helia** [40]. Esto implica que es nuestra responsabilidad definir cómo se comunican los nodos, seleccionando explícitamente qué protocolos de transporte utilizar según el entorno en el que se ejecutan.

LibP2P es una biblioteca modular diseñada para construir redes *peer-to-peer*. No define una única forma de conexión, sino que ofrece un conjunto de componentes intercambiables —como transporte, cifrado, multiplexación o descubrimiento de pares— que pueden combinarse según las necesidades del entorno. Esta flexibilidad es clave para nuestra infraestructura, ya que nos permite adaptar el comportamiento de los nodos a las restricciones del entorno web o aprovechar las capacidades completas de un proceso independiente.

LibP2P proporciona distintos protocolos de transporte que permiten que dos nodos mantengan una conexión persistente. Cada nodo puede configurarse con múltiples protocolos, y la elección depende de dónde se ejecuta ese nodo. En nuestra infraestructura, distinguimos entre:

- **Nodos independientes**, que corren en entornos como node.js
- **Nodos web**, que se ejecutan dentro de un navegador.

Los nodos independientes utilizan **TCP** como protocolo principal. Es estable, ampliamente soportado y no presenta restricciones técnicas en entornos controlados. Este protocolo permite que los nodos colaboradores se comuniquen entre sí directamente.

Por otro lado, los nodos web presentan mayores restricciones. Los navegadores modernos no permiten conexiones TCP o UDP directas por razones de seguridad, y limitan las conexiones a protocolos seguros que cumplan con políticas de contexto seguro (*Secure Context*), como HTTPS.

Para resolver esta limitación, la infraestructura utiliza **WebRTC-Direct**, un protocolo de transporte compatible con navegadores que permite conexiones directas entre un nodo web y un nodo independiente. **WebRTC**[88] es una tecnología estándar para aplicaciones en tiempo real, como llamadas de video o intercambio de archivos. LibP2P lo adapta para lograr una conexión peer-to-peer entre nodos, sin depender de certificados ni servidores intermediarios.

Gracias a esta combinación de protocolos —**TCP para nodos independientes y WebRTC-Direct para nodos web**—, se logra una red completamente funcional en la que cualquier nodo puede conectarse con otros, incluso desde el navegador, sin comprometer la descentralización de la arquitectura.

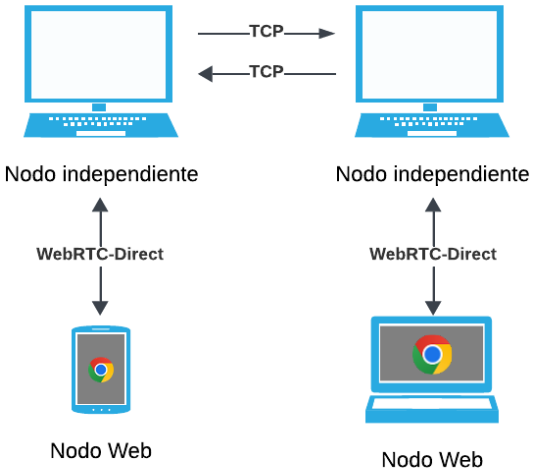

Figura 14: Topología de conexiones entre nodos independientes y nodos web.

Descubrimiento de colaboradores En la infraestructura de AstraDB, además de definir los protocolos de transporte y la forma en que los nodos se conectan, es fundamental abordar el mecanismo mediante el cual un nodo identifica a qué otros nodos debe conectarse para sincronizar y obtener la base de datos, es decir, cómo se realiza el descubrimiento de colaboradores.

Para preservar la naturaleza descentralizada del sistema, esta función se implementa aprovechando el modelo de descubrimiento de contenido que utiliza IPFS. IPFS se basa en un esquema de *content addressing* [24], donde los datos son identificados y accedidos a través de un identificador único basado en su contenido, conocido como *Content Identifier* (CID), en lugar de su ubicación física.

La localización del contenido en IPFS se efectúa mediante una *Distributed Hash Table* (DHT), que funciona como una base de datos distribuida. Cada nodo mantiene una porción del índice global que asocia cada CID con los nodos proveedores de dicho contenido. Cuando un nodo desea acceder a un contenido específico, consulta la DHT para identificar los proveedores activos correspondientes [29].

Este mecanismo se adopta directamente en AstraDB para el descubrimiento de colaboradores. Al considerar que el CID representa la base de datos —derivada de su nombre identificador— y que los colaboradores son los nodos que proveen ese CID, un nodo interesado realiza una consulta en la DHT para obtener una lista de nodos activos replicando la base. De esta forma, se establece un enfoque completamente descentralizado en el que los colaboradores anuncian su disponibilidad y los nodos nuevos pueden descubrirlos y sincronizarse sin depender de nodos preconfigurados ni puntos centrales.

Identidad de los usuarios En AstraDB, los usuarios no se identifican mediante nombres de usuario centralizados, sino a través de un esquema criptográfico basado en claves públicas y privadas, gestionado mediante las identidades de OrbitDB.

Si al iniciar un nodo de AstraDB no se proporciona una clave privada, se genera automáticamente una nueva identidad. La clave privada asociada puede luego ser exportada y almacenada, permitiendo preservar la identidad del nodo entre sesiones.

Arquitectura La infraestructura de AstraDB se sustenta en los conceptos y tecnologías detallados en las secciones anteriores, integrando el almacenamiento descentralizado mediante IPFS y OrbitDB, junto con el manejo de conexiones peer-to-peer a través de LibP2P. Esta base técnica

permite construir una arquitectura modular que separa claramente el manejo de datos y el manejo de conexiones.

Dicha arquitectura está compuesta por dos componentes principales que operan en paralelo y de manera independiente: el *Key Repository* y el *Connection Manager*.

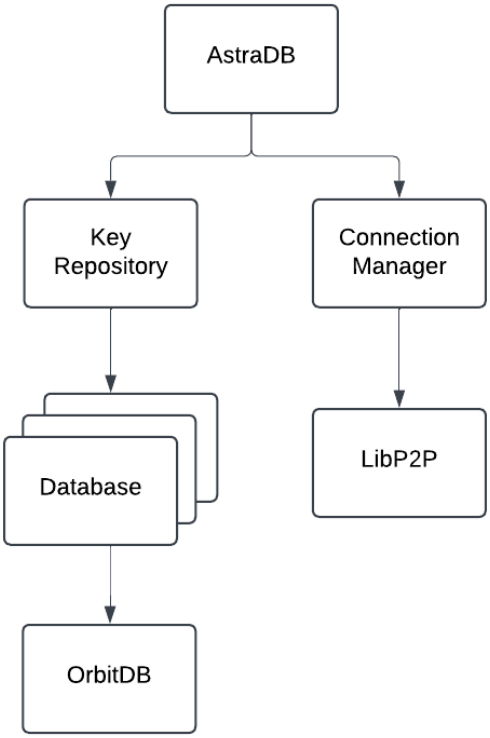

Figura 15: Arquitectura de AstraDB.

Key Repository El *Key Repository* es el módulo encargado de la gestión integral de los datos dentro de la infraestructura. Su función principal consiste en mantener el registro de todas las claves existentes, así como efectuar las modificaciones necesarias sobre ellas.

Al iniciarse, este componente crea la base de datos central a partir del nombre recibido como parámetro y procede a sincronizarla con los colaboradores disponibles. Si no se logra establecer conexión con ningún colaborador, se asume que se trata de una base de datos nueva, y el módulo continúa su funcionamiento en modo autónomo.

En nodos configurados como colaboradores, al recibir actualizaciones sobre nuevas claves desde la base de datos central, el *Key Repository* abre y almacena localmente la base de datos correspondiente a cada clave. Este proceso garantiza la persistencia de las claves y permite que otros usuarios accedan a ellas mediante sincronización.

Todas las bases de datos gestionadas por este módulo —tanto la central como las asociadas a cada clave— están representadas mediante una abstracción denominada *Database*, la cual encapsula toda la interacción con OrbitDB.

Durante la inicialización de una instancia de *Database*, se debe definir si la base de datos requiere sincronización. En caso afirmativo, el sistema espera conectarse con algún colaborador para obtener todas las actualizaciones disponibles. Una vez sincronizada, la base queda habilitada para su uso, incluyendo la incorporación de nuevo contenido y la recuperación completa del mismo.

Además, toda actualización recibida en una base de datos gestionada por este módulo genera automáticamente un evento que puede ser escuchado por otros componentes del sistema. Esto

permite implementar funcionalidades reactivas, como interfaces en tiempo real, facilitando la integración de lógica adicional basada en los cambios del estado distribuido.

Connection Manager El *Connection Manager* se encarga del manejo de conexiones entre nodos, incluyendo la búsqueda y establecimiento de conexiones con colaboradores que proveen la base de datos, así como la publicación del nodo como proveedor cuando este actúa como colaborador.

Al iniciarse, el módulo construye el CID correspondiente a la base de datos central. Para ello, utiliza **Helia** [40], la implementación de IPFS en JavaScript, para calcular el identificador a partir del nombre recibido. Este CID es común a todas las instancias de AstraDB que utilizan dicho nombre, y permite que los nodos se identifiquen y sincronicen de forma coherente.

Luego, se inicializan tres servicios que operan concurrentemente durante la ejecución del sistema: *SearchForProviders*, *ProvideDB* y *ReconnectToProviders*.

El servicio *SearchForProviders* realiza búsquedas continuas de nuevos proveedores de la base de datos utilizando la DHT de IPFS mediante LibP2P. Cuando encuentra uno, intenta establecer conexión para iniciar la sincronización.

El servicio *ProvideDB*, activo únicamente en nodos colaboradores, anuncia en la DHT que el nodo puede proveer la base de datos, facilitando su descubrimiento por parte de otros.

Finalmente, *ReconnectToProviders* mantiene un listado de proveedores previamente conectados y realiza intentos periódicos de reconexión, asegurando la continuidad en caso de fallos temporales.

Estos servicios permiten que los nodos se descubran entre sí, compartan información de manera descentralizada y mantengan la red resiliente ante desconexiones o caídas parciales.

Implementación en los casos de uso La arquitectura y los mecanismos de AstraDB fueron puestos a prueba en dos casos de uso representativos: el repositorio de conocimiento y el mensajero en tiempo real.

En el caso del repositorio de conocimiento, las claves representaron los artículos de la wiki, mientras que los valores correspondieron a las modificaciones realizadas sobre cada artículo. De esta forma, fue posible reconstruir el estado actual de un artículo a partir de toda su historia de cambios almacenada.

Para el mensajero en tiempo real, cada clave representó un chat específico y los valores fueron los mensajes enviados en dicho chat. El sistema permitió el inicio de sesión de usuarios mediante la provisión de una clave privada, y habilitó la escucha en tiempo real de las actualizaciones en un chat determinado, garantizando así la recepción inmediata de nuevos mensajes.

Ambos casos de uso lograron, en su implementación, abstraerse del manejo directo del ecosistema subyacente —como IPFS, OrbitDB y LibP2P— gracias a los mecanismos provistos por AstraDB. Esta capa de abstracción permitió que los desarrollos se concentraran exclusivamente en la lógica específica de cada aplicación, sin necesidad de centrarse en detalles propios de la infraestructura distribuida.

Como resultado, no solo se cumplió con los requisitos funcionales de ambos casos, sino que se sentaron las bases para extender el sistema a nuevos escenarios. Siempre que las entidades de un caso de uso puedan representarse mediante claves y su evolución como una secuencia de eventos, AstraDB proporciona un entorno sólido y reutilizable para construir aplicaciones distribuidas de forma simple y modular.

9.2.3. Colaboración

Se creó una herramienta para facilitar la colaboración a nuestra principal interfaz de usuario. El nodo es capaz de fijar los contenidos de **Astrawiki-web** —nuestra interfaz de usuario—, y también mantener todas las bases de datos utilizadas por **Astrawiki** y **Astrachat** en la capa de aplicación. Para lograr esto, se diseñó un sistema con cuatro contenedores, cada uno asignado a uno de los tres casos de uso.

Para el *pinning* del sitio web en si, se utilizó el contenedor de **IPFS Cluster** con la instrucción de colaborar con el clúster dado por parámetro. Este parámetro es la dirección IPFS del archivo `service.json` que identifica a un clúster. Además, IPFS Cluster requiere de un contenedor de **Kubo** para manejar el manejo de archivos en la red de IPFS.

Para **Astrawiki**, se hizo uso del contenedor ofrecido por Astrawiki CLI, interfaz de usuario que se verá más adelante. Con este contenedor, se puede actuar como colaborador fácilmente sin necesidad de interactuar con él.

Por último, **Astrachat** se levanta con un simple proyecto de Node.js que crea un nodo colaborativo y lo ejecuta en segundo plano.

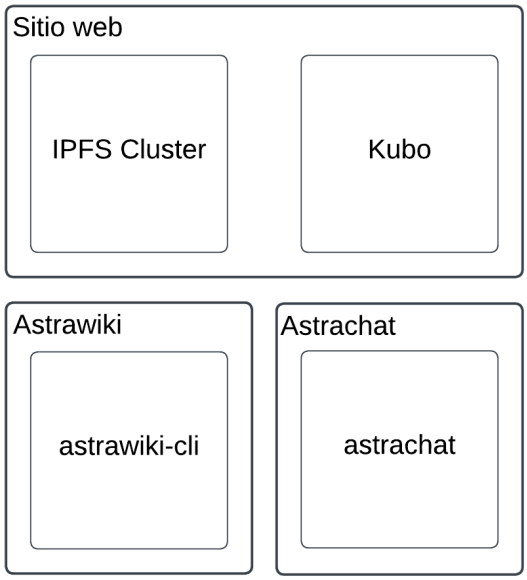

Figura 16: Arquitectura general del nodo colaborador

Cada uno de estos casos se puede habilitar o deshabilitar desde un archivo de configuración `.env`, lo que permite al usuario elegir en qué parte del proyecto desea colaborar.

Además, el sistema de colaboración fue diseñado con un enfoque flexible: su configuración puede adaptarse fácilmente para apuntar a otras instancias de los distintos casos de uso, lo que permite que el nodo colaborador se integre sin dificultad a diferentes aplicaciones de **Astrawiki** o **Astrachat**.

9.3. Blockchain

Para el ecosistema de blockchain se decidió utilizar la red de Ethereum [92], al ser una blockchain popular y muy utilizada para el desarrollo de aplicaciones descentralizadas nos permite demostrar y comparar los casos de uso contra nuestra solución en IPFS.

9.3.1. Swarm

Existen varias soluciones de almacenamiento descentralizado dentro del ecosistema blockchain, en este caso, para el desarrollo del sitio web estático optamos por Swarm al estar basado en una *sidechain* de Ethereum.

Feed El contenido que se publica en Swarm tiene asociado un CID, esto lo hace inmutable. Para permitir cambios manteniendo la inmutabilidad del contenido en Swarm existen los *feeds* que

funcionan de manera similar a los nombres de IPNS en IPFS. Es un puntero, con CID fijo, a un archivo. Esto permite actualizar el archivo al que apunta un *feed* manteniendo un punto de entrada fijo al sitio web.

Postage stamps Esta es la forma de pagar por el uso del almacenamiento en la red de Swarm. Posee un *time-to-live* (TTL) que es calculado en base a los valores de *depth* y *amount* indicados al momento de la creación del *stamp* [85].

Despliegue Para el despliegue del *front-end* es necesario primero comprar un *postage stamp* con la moneda BZZ y luego con la herramienta `swarm-cli` [84] se publica el sitio web indicando el *feed* correspondiente. Durante el desarrollo encontramos que no existe un *gateway* público que apunte a la *testnet* de Sepolia y nos permita probar el despliegue sin necesidad de utilizar dinero real. Es por esto que se levantó un nodo de Swarm configurado para que apunte a la *testnet* de Sepolia y un *gateway* que utilice este nodo mediante la herramienta `gateway-proxy` [38] que provee el equipo de Swarm.

9.3.2. Ethereum

Para los casos de uso del repositorio de conocimiento y el mensajero en tiempo real necesitamos una herramienta que funcione de manera *read-write* y como Swarm solamente se encarga de archivos estáticos buscamos alguna alternativa dentro del ecosistema blockchain. Para esto terminamos usando Ethereum.

Figura 17: *Smart contracts* que intervienen en el repositorio de conocimiento

Ambos casos de uso resultaron muy similares en su resolución, haciendo uso del patrón de diseño *Factory*. Existe un *smart contract Factory* que crea otros *smart contracts* (Artículo o Chat, según el caso de uso).

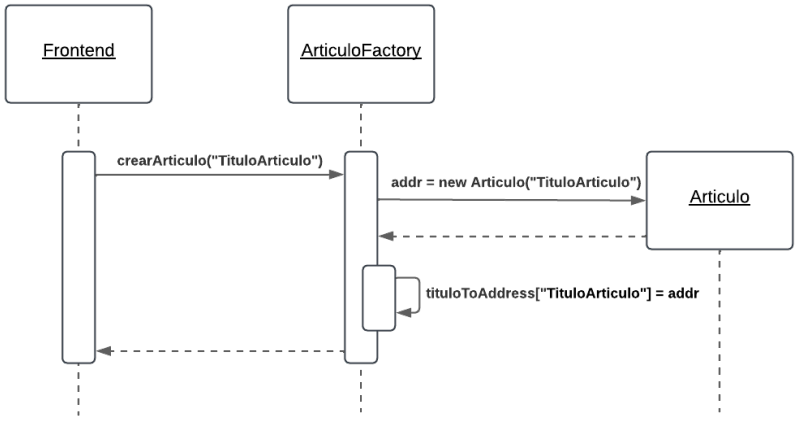

Figura 18: Creación de un artículo

De esta manera el *Factory* tiene un *mapping* con todos los artículos creados y las direcciones correspondientes para accederlos. Si se quisiera acceder a un Artículo en particular primero se tiene que consultar al *Factory* para obtener la dirección del mismo y, como cada artículo es un *smart contract* en sí mismo, se puede consultar o modificar su contenido directamente interactuando con el Artículo en particular como se puede ver en la Figura 19.

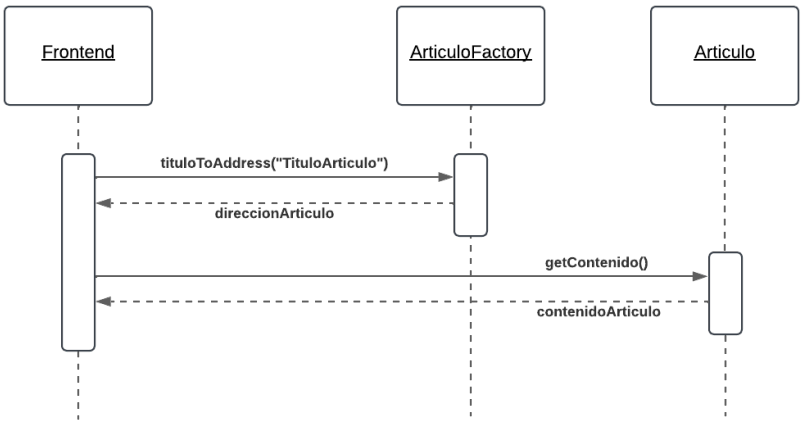

Figura 19: Obtención del contenido de un artículo

La principal diferencia entre el repositorio de conocimiento y el mensajero en tiempo real está en que los mensajes del mensajero tienen que ser vistos por los demás usuarios que participan de la conversación en el momento que se envían. Esto no es estrictamente necesario en el repositorio de conocimiento pero sí lo es en el mensajero.

Para afrontar este requisito se utilizaron los eventos de Solidity (el lenguaje de programación en el que se desarrollan los *smart contract* de Ethereum). Funciona de la siguiente manera, al momento de enviar un mensaje se emite un evento. Este evento se recibe en un *listener* que fue previamente inicializado al instante previo de haber obtenido el Chat en el *front-end*. Al recibir este evento el *front-end* puede actualizar la pantalla mostrando el mensaje nuevo sin necesidad de obtener todos los mensajes.

Por otro lado, para el mensajero en tiempo real necesitamos una manera de identificar a cada usuario. Para esto se hizo uso de las *wallets*. Cada usuario se identifica utilizando su *wallet*, que tiene una clave pública, que pasa a ser el identificador del usuario, y una clave privada la cuál es necesaria para firmar transacciones en nombre del usuario, que en nuestro caso funciona a modo de contraseña. Además, para que la lectura de las conversaciones sean más usables, se agregó la posibilidad de generar un nombre de usuario asociado al identificador del mismo. A este nombre de usuario lo llamamos alias y es único para todos los Chats asociados a un mismo *ChatFactory*. Una vez el usuario se conecta con su *wallet*, puede elegir un alias y cambiarlo cuando desee siempre y cuando no exista actualmente algún otro usuario con ese mismo alias.

Finalmente, nos queda la funcionalidad de que un usuario pueda responder a otro mensaje. Primero necesitamos una manera de identificar a cada mensaje de manera unívoca. El identificador de cada mensaje se genera *hasheando el timestamp* del bloque, el identificador del emisor y la cantidad de mensajes en el chat en el momento que se envía. Luego, cuando se responde a otro mensaje se almacena el identificador del mensaje al que se está respondiendo (el mensaje padre) dentro de la estructura del mensaje que se está enviando. Todo esto se resuelve dentro del *smart contract* del Chat correspondiente.

9.4. Hyphanet

A continuación se describen las distintas características del ecosistema hasta lo que se llegó a investigar.

9.4.1. Arquitectura

La plataforma se basa en conexiones por nodos que gestionan la información que hay en la red. Estos nodos se conectan a una red abierta llamada *Opennet* en la cual las conexiones se hacen con nodos de cualquier persona de cualquier parte del mundo. Por otro lado es posible configurar el nodo para que se conecte sólo con aquellas personas que uno conozca, creando así una red "privada" (o *darknet*) en la que sólo personas de confianza puedan conectarse.

El contenido que se publica en los nodos permanece de forma encriptada y repartido en varias partes por distintos nodos. Siempre que un archivo sea solicitado el mismo será cacheado en los distintos nodos que lo soliciten. Para acceder al contenido es necesario correr un nodo y conectarse a la red de Hyphanet ya que no es posible conectarse desde la red de internet convencional.

No es posible "borrar" un archivo como tampoco es posible guardarlo a voluntad. Si un archivo no es suficientemente solicitado eventualmente no se puede recuperar más. Los nodos tampoco pueden elegir qué contenido (partes de un archivo) guardar o no ya que los mismos están encriptados. Si un archivo es subido por un nodo que no está disponible en este momento el archivo no se pierde porque ya fue distribuido por los demás nodos que estaban conectados a él.

9.4.2. Plugins

Es posible crear aplicaciones de comunicación con los llamados *plugins*. Estos *plugins* deben ser hechos en el lenguaje Java (o al menos el `main` debe estarlo) por decisión de diseño (alegando que Java es "más seguro"). No están aislados del sistema *host* por lo que pueden acceder a toda la información que quieran. Se deben compilar y proveer el `.jar` correspondiente y cada usuario que quiera utilizarlo debe instalarlo en su respectivo nodo.

Entre los plugins más usados en el ecosistema podemos nombrar:

WebOfTrust Se autodenomina un *spam filter* pudiendo puntuar (*trust values*) a cada usuario de forma que los que tienen puntaje muy bajo son catalogados como spammers y cualquier contenido que los involucre será filtrado [51].

Sone Red social similar a Facebook en el que se pueden subir imágenes, comentar, conectarse con otras personas. Usa WebOfTrust para identificar a cada usuario [50].

Freemail Un servicio de email dentro de Hyphanet que también depende de WebOfTrust [47].

Freetalk Sistema de foro [48].

No hay una receta para implementar estos plugins ya que la guía [52] que existe está incompleta y hace años que no se actualiza (lo mismo pasa con los plugins en sí, llevan años sin actualizarse).

La documentación es prácticamente nula y cada *plugin* hace uso de la librería de *Freenet* de una forma distinta lo cual hace difícil saber cuál es la forma correcta (de haberla) para crear un *plugin* desde cero.

Toda información que deba guardarse se debe hacer en una base de datos local administrada por el *plugin* (ya sea usando un archivo o una librería como podría ser *sqlite* [82]) ya que no existe una base de datos distribuida en el ecosistema que lo facilite. Esto hace que cada nodo tenga la información sólo de aquellos nodos con los que interactúa, mientras más lo haga más datos va a

tener que persistir. Además es necesario que el *plugin* tenga forma de garantizar la integridad de los datos sino podrían ser fácilmente manipulados por algún nodo generando inconsistencias para la aplicación (que a su vez debería poder manejar).

9.4.3. Sitio web estático

Hyphanet posee un software propio para poder agregar sitios llamado *jSite* [49]. Con el mismo, basta con seguir las instrucciones en la documentación oficial [46]. Una vez finalizada la creación del sitio, el programa devolverá un *hash* el cual será necesario para poder acceder al sitio.

Caso de uso Se logró levantar un sitio web estático que sólo contenía un HTML con un `"Hello World"` utilizando *jSite* [49] en el ecosistema.

9.4.4. Otros casos de uso

Debido a la escasa documentación y falta de estándares para desarrollar aplicaciones en el ecosistema, no se prosiguió con los demás casos de uso y se optó por investigar el ecosistema *Freenet*.

9.5. Freenet

Siguiendo las bases de Hyphanet, y a diferencia de IPFS, Freenet busca ser una *computadora distribuida* donde cada peer es capaz de ejecutar código que contenga estado el cual tendrá eventual consistencia con los demás *peers*. Parecido a su antecesor, un contrato puede cachearse en varios *peers* si este es lo suficientemente popular y dejará de cachearse si deja de serlo.

9.5.1. Arquitectura

Key-value Freenet es un *global key-value store* que se basa en la idea del *small-world routing* [81] para la descentralización y escalabilidad. Las *keys* son código WebAssembly en dónde se especifican:

- Qué valores están permitidos en la *key*.
- Bajo qué circunstancias el valor puede ser modificado.
- Cómo se puede sincronizar el valor eficientemente entre los *peers* de la red.

Contracts La base de la comunicación de las aplicaciones distribuidas son los *contracts*. Estos *contracts* son código Rust compilado a WebAssembly donde una clase debe cumplir con la interfaz del contrato (`ContractInterface`). Este es encargado de mantener la consistencia del estado de la aplicación en los distintos *peers*. Está pensado para que la actualización sea eficiente de modo que los *contracts* se actualizan en base a las diferencias.

Contracts vs Smart Contracts Los *contracts* de Freenet poseen ciertas similitudes y diferencias con los *smart contracts* de Blockhain.

	Contracts (Freenet)	**Smart Contracts (Blockchain)**
¿Descentralizado?	Si	Si
¿Mantiene estado?	Si	Si
¿Puede ejecutar código arbitratiamente?	Si (limitación local)	Si (limitación global)
¿Se distribuye el estado entre distintas instancias?	Si	No
¿Pago?	No	Si

Tabla 1: Comparativa entre Contracts de Freenet y Smart Contracts de Blockchain

Comunicación Al momento de probar el ecosistema, una aplicación podía comunicarse con un contrato a través de un *web socket*. De esta forma se busca mantener el estado actualizado en la aplicación local ante un eventual cambio hecho por otro *peer*.

9.5.2. Diferencias con Hyphanet

Funcionalidad Hyphanet es un disco duro descentralizado mientras que Freenet busca ser una computadora descentralizada.

Interacción en tiempo real Freenet permite a los usuarios subscribirse a los datos y ser notificado inmediatamente ante algún cambio del mismo. Esencial para aplicaciones de chat o interacción en tiempo real.

Lenguaje de programación Hyphanet fue desarrollado en Java. Freenet está implementado en Rust haciéndolo más eficiente y pudiéndose integrar mejor a los distintos sistemas operativos (Windows, Mac, Android, etc).

Transparencia Freenet está pensado como un posible reemplazo a World Wide Web.

Anonimato La versión anterior fue diseñada con foco en el anonimato. La nueva versión no ofrece esta opción *out-of-the-box* sin embargo es posible crear un sistema por encima que provea cierta anonimidad.

9.5.3. Baja del ecosistema

Debido a las constantes modificaciones en la API del ecosistema y la falta de actualización de la documentación por estar en pleno desarrollo, tuvimos que optar por descartar el ecosistema, por lo que no se llegaron a implementar ninguno de los casos de uso propuestos.

9.6. Interfaces de usuario

Una de las decisiones arquitectónicas fundamentales del proyecto fue desarrollar cada caso de uso como un paquete independiente, encapsulando en cada uno la lógica específica necesaria para operar sobre su respectivo ecosistema (IPFS o Ethereum). Esta modularidad permitió abstraer la complejidad técnica detrás de cada solución y exponer interfaces simples y reutilizables, facilitando su integración desde cualquier aplicación cliente, especialmente desde el navegador.

Gracias a este enfoque, fue posible construir múltiples interfaces gráficas que, sin necesidad de replicar lógica de negocio, pueden interactuar con los diferentes módulos según el contexto o ecosistema seleccionado. Esta versatilidad permitió no solo demostrar cada caso de uso de forma aislada, sino también combinarlos en un único entorno integrado.

Astrawiki-web Es la principal interfaz de usuario, y acumula todos los casos de uso creados en una única aplicación web, demostrando la capacidad de utilizar estas herramientas en conjunto, en un entorno web, y de manera intercambiable. Se le permite al usuario escoger el ecosistema al que se desea conectar, pudiendo intercambiar ecosistemas libremente, a forma de comparar y evaluar la funcionalidad desarrollada en este proyecto.

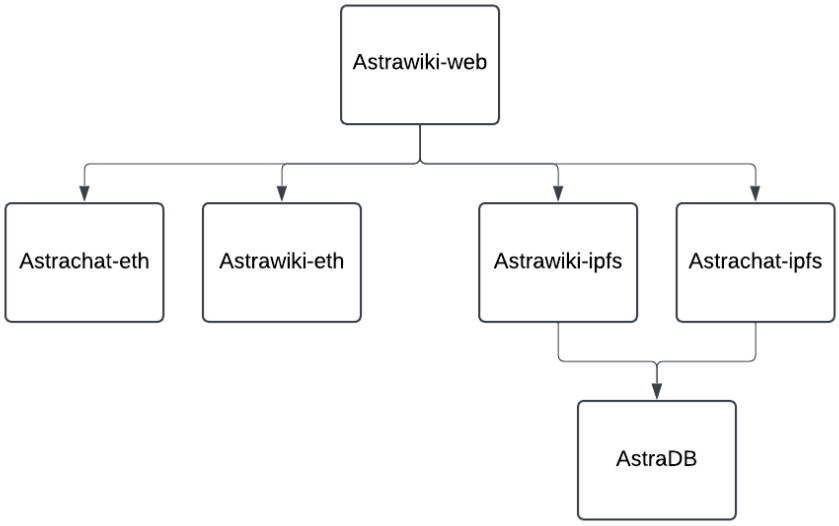

Figura 20: Diagrama de componentes de Astrawiki-web

Representa un repositorio de conocimiento, con discusiones en cada articulo, lo que se asemeja a repositorios como Wikipedia. Además, se alojan en las redes descentralizadas evaluadas, tanto en IPFS como en Swarm, accesibles desde el navegador.

Funcionalidad Desde la página principal es posible seleccionar entre tres ecosistemas distintos:

- Example Server
- IPFS
- Blockchain

Una vez seleccionado el ecosistema se puede acceder a la barra de búsqueda para buscar el artículo deseado o bien se puede crear un artículo.

En la creación de un artículo (Figura 22), es posible darle un nombre y una descripción la cual puede ser escrita en Markdown para darle estilo. Una vez dentro de un artículo (Figura 23), a la derecha se puede acceder al historial de versiones (donde se puede visualizar una versión específica del artículo), editar el artículo (sólo en contenido puede ser modificado) y también al chat del artículo.

Cada artículo posee un chat en tiempo real (Figura 24) que actúa como discusión para la mejora o modificación del mismo. Dentro del chat es posible enviar mensajes y responder a mensajes anteriores desde un alias configurado por el usuario. Para mantener la identidad dentro de los chats, se puede ingresar con la misma identificación.

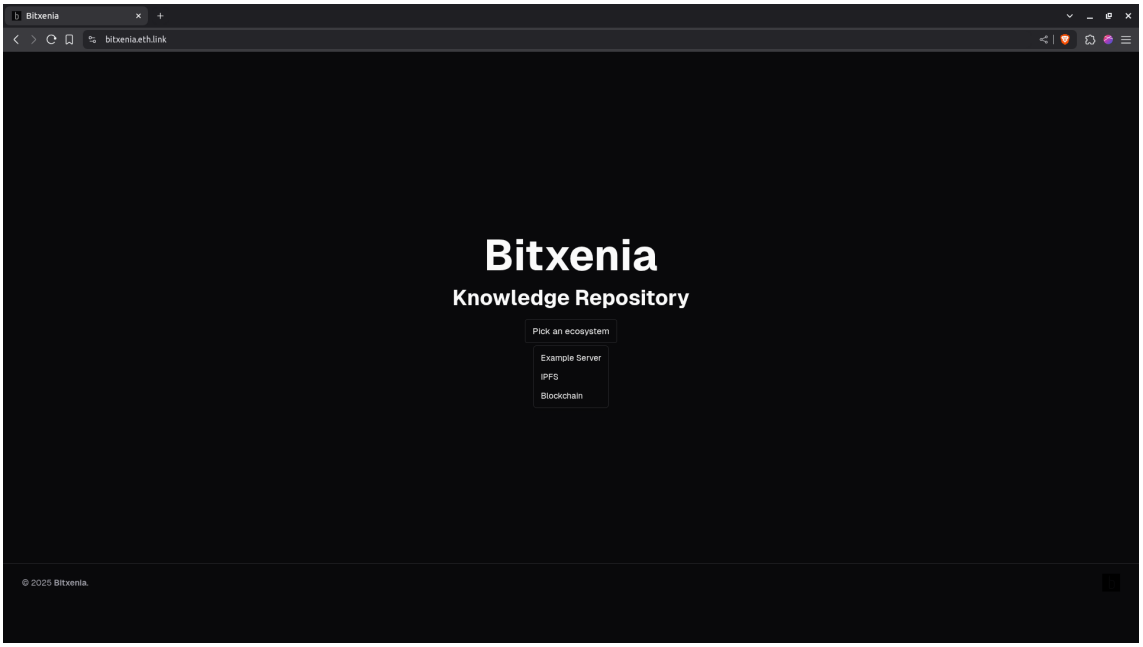

Figura 21: Página principal de Astraweb

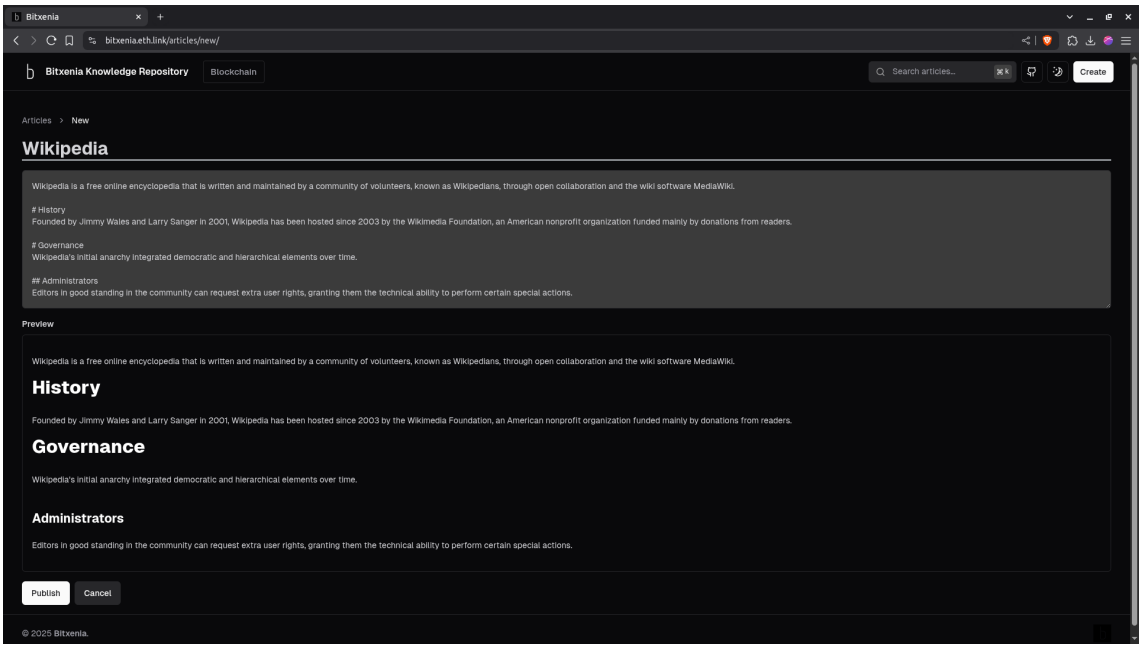

Figura 22: Creación de un artículo

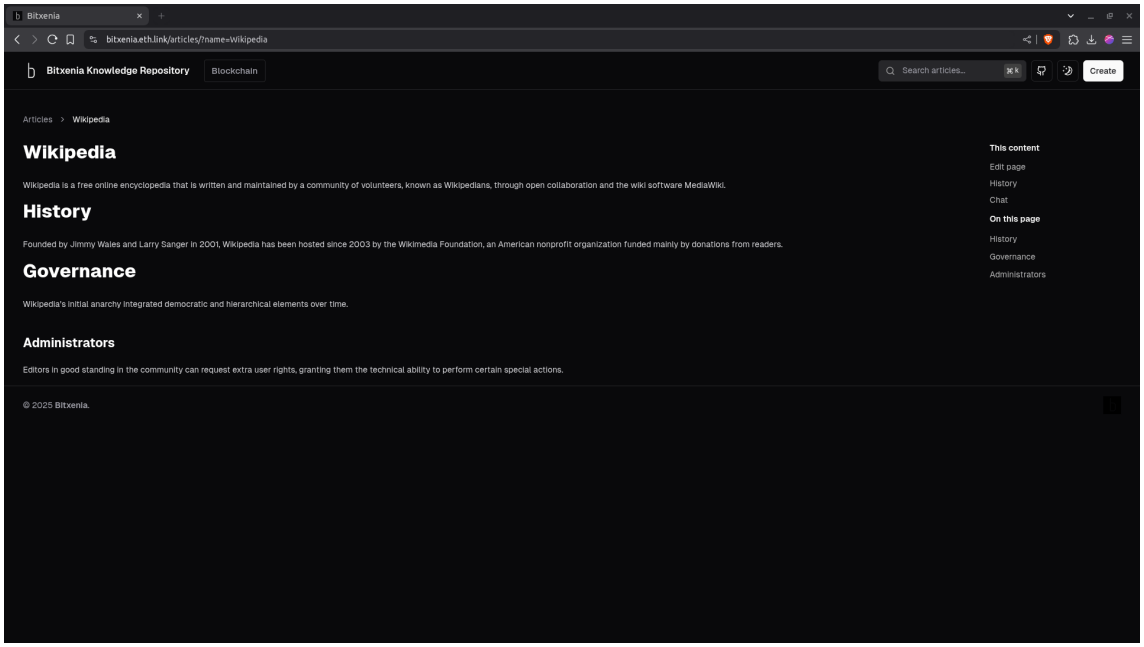

Figura 23: Página de un artículo

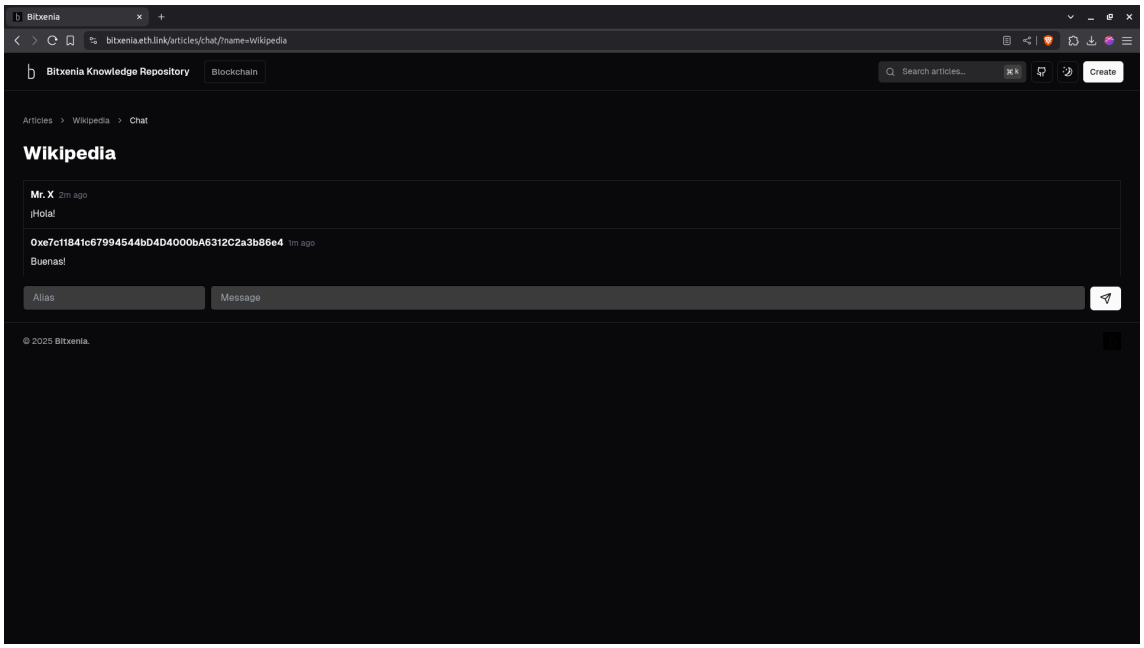

Figura 24: Chat de un artículo

Tecnología Se utilizó `React` [75] y `Next.js` [76] como *frameworks* para la creación de la aplicación web, basándonos en una plantilla llamada `rubix-documents` [79]. El código fue escrito en `Typescript`.

Servidor ejemplo Se desarrolló una solución centralizada como tercer ecosistema para este *front-end*, con el propósito de paralelizar el desarrollo y los distintos paquetes de cada ecosistema. Debido a que se definió una interfaz común tanto para el repositorio de conocimiento como para el mensajero en tiempo real, se logró avanzar con el *front-end* haciendo pruebas manuales con este

servidor.

El servidor fue creado en `Node.js` con `Express.js` como framework para interactuar con *requests* de HTTP.

Limitaciones Debido a la falta de un servidor tradicional para ofrecer el contenido, el uso de *Server Components* o componentes de servidor [80] no era posible. Esto implicó modificar ampliamente la plantilla utilizada para descartar este tipo de componentes en favor de aquellos que pueden ser compilados y luego utilizados por el cliente sin interacción con un servidor.

Astrawiki CLI Front-end de terminal, desarrollado para el caso de uso del repositorio de conocimiento y para el ecosistema de IPFS en específico. Cuenta con todas las funcionalidades del repositorio de conocimiento, como crear, editar y ver artículos, consultar versiones pasadas, e incluso colaborar, lo cual se verá en el apartado de IPFS. Además, cuenta con un contenedor `Docker` publicado, con el fin de fácilmente iniciar un nodo sin necesitar una instalación de `Node` y demás dependencias. Funciona como un `daemon` [25], es decir, se inicia y funciona en segundo plano hasta que se indique lo contrario.

Su propósito es, por un lado demostrar la versatilidad del modelo de paquetes utilizado para crear distintos *front-ends*. Y por otro lado, el contenedor es útil para el nodo colaborador desarrollado para IPFS visto previamente.

Arquitectura Se compone de un cliente, el cuál se inicia con cada comando (`start`, `add`, `get`, `edit`, `list`, etc.) y un servidor que se ejecuta por detrás, el cual inicia la instancia del repositorio de conocimiento de IPFS y utiliza su API cuando recibe `requests HTTP`.

Figura 25: Ejemplo de uso de `astrawiki-cli`

Astrachat CLI Frontend de terminal, desarrollado para el caso de uso de mensajero en tiempo real y para el ecosistema de Ethereum.

Tecnología Se utilizó `React` [75] y `Ink` [53] como *frameworks* para crear la interfaz de este *front-end*.

Funcionalidad Esta implementación permite crear un chat, unirse a un chat existente, enviar y leer los (últimos) mensajes. No provee de la opción de modificar el alias ni tampoco de responder a otro mensaje.

Figura 26: Página principal de `astrachat-cli`

Limitaciones Debido a que se ejecuta en la terminal no es posible hacer uso de una *wallet* como Metamask [63] ya que la misma sólo funciona en web. Debido a esto, se *hardcodearon* algunas de las *wallets* que provee Hardhat [39].

10. Metodología aplicada

La metodología aplicada para la gestión del proyecto fue una versión simplificada de Scrum. El desarrollo se dividió en *sprints* semanales para los cuales utilizamos un tablero *Kanban* en Github Projects donde se fueron agregando las tareas a realizar para cada caso de uso y ecosistema. Se realizaron reuniones semanales fijas que se usaron como punto de control, donde se revisó lo hecho durante la semana y definimos pasos a seguir para las siguientes. También nos fue útil para detectar posibles ajustes o cambios de rumbo que fueron surgiendo a lo largo del trabajo. Al finalizar cada una de estas reuniones realizamos minutas que nos sirvieron para resumir lo tratado en cada una y tener en claro los pasos a seguir después de las mismas.

Durante el descubrimiento de las funcionalidades de cada caso de uso realizamos *User Story Mappings* (USM) para organizar las tareas a realizar.

Los distintos artefactos que fueron surgiendo durante el desarrollo del trabajo fueron almacenados en un Google Drive compartido entre todo el equipo. Dentro del mismo se pueden encontrar: minutas de reuniones, cronogramas, USM, entre otros.

Para el control de versiones del código se creó una organización en Github en la que se fueron creando repositorios para los distintos paquetes que integraron el trabajo.

La modalidad fue virtual y asincrónica. Nos mantuvimos en constante comunicación a través de un servidor de Discord y también se realizaron sesiones de *pair* y *mob-programming* en distintas ocasiones.

11. Experimentación y/o validación

En base a los casos de uso implementados se realizó un análisis cualitativo y cuantitativo de cada ecosistema, con el objetivo de compararlos y ver las ventajas y desventajas de cada uno, teniendo en cuenta las siguientes categorías:

- **Costos**: ¿Cuánto nos cuesta desplegar y mantener un servicio en cada ecosistema? ¿Un usuario final necesita efectuar algún costo monetario para el uso de la aplicación?
- **Experiencia de desarrollo**: ¿Qué tan fácil es desplegar en cada ecosistema? ¿Existen herramientas y documentación necesario que facilite el desarrollo?
- **Aplicabilidad al caso de uso**: ¿Qué tan viable es crear una aplicación comunitaria para cada uno de estos ecosistemas?
- **Performance**: ¿Cuánto tiempo tarda la creación y obtención de elementos en cada aplicación?

11.1. Propiedades utilizadas para las métricas

Para las distintas pruebas cuantitativas realizadas, se establecieron tamaños fijos según el tipo de contenido evaluado.

11.1.1. Sitio Web Estático

El sitio web estático se clasificó por el tamaño total del contenido alojado:

- *1KiB-50files*: 50 archivos que en conjunto tienen un tamaño de 1KiB
- *25KiB-50files*: 50 archivos que en conjunto tienen un tamaño de 25KiB
- *50KiB-1files*: 1 archivo que tiene un tamaño de 50KiB
- *50KiB-10files*: 10 archivos que en conjunto tienen un tamaño de 10KiB
- *50KiB-25files*: 25 archivos que en conjunto tienen un tamaño de 25KiB
- *50KiB-50files*: 50 archivos que en conjunto tienen un tamaño de 50KiB

11.1.2. Repositorio de conocimiento

Los artículos se separaron en tres clasificaciones, según su longitud en bytes:

- *short*: artículo de 5000 bytes.
- *medium*: artículo de 10000 bytes.
- *large*: artículo de 50000 bytes.

11.1.3. Mensajero en tiempo real

Para la medición del envío y recepción de mensajes, se consideró tres tipos de mensajes:

- *short*: mensaje de 1 palabra. "*Lorem*"
- *medium*: mensaje de 10 palabras. "*Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam elementum.*"
- *large*: mensaje de 30 palabras. "*Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam ex risus, porttitor sed lacus id, egestas lobortis purus. Curabitur consectetur metus ut est vehicula egestas. Class aptent taciti sociosqu ad.*"

El tamaño de las palabras elegidas representa el largo de los mensajes más comunes que suelen enviarse en un chat.

A partir de los datos obtenidos se calcularon el máximo (**Max**), mínimo (**Min**), la media (**Mean**), el desvío estándar (**Std**) y la mediana (**Median**) de cada muestra.

11.2. IPFS

11.2.1. Costos

En las soluciones de IPFS presentadas, se puede hacer una distinción entre colaboradores y usuarios finales. Si bien ambos ejecutan un nodo propio más en la red, el colaborador tiene más responsabilidades y por lo tanto, utiliza más almacenamiento y cómputo disponible. Como un usuario final únicamente se conecta a la red sin necesidad de almacenar todo el contenido de una aplicación o proveer dicho contenido a los demás nodos, su costo energético es comparable con el de un usuario final utilizando un servicio de Ethereum. No hay análisis conclusos al respecto, pero como trabajo futuro puede ser muy útil para conocer la eficiencia de la red en comparación con una solución de tipo Blockchain.

11.2.2. Experiencia de desarrollo

La experiencia de desarrollo dentro del ecosistema de IPFS resultó desafiante, principalmente debido a que se trata de una tecnología emergente y todavía poco adoptada en entornos de producción. Esto se tradujo en una falta de madurez en algunas herramientas clave y en una documentación a veces limitada o desactualizada.

En nuestro caso, se eligió trabajar con herramientas del ecosistema JavaScript, dado que OrbitDB está diseñado para este entorno y era necesario asegurar compatibilidad con navegadores web. No obstante, JavaScript no representa el entorno más maduro ni prioritario dentro de IPFS y libp2p, lo cual implicó enfrentar una serie de desafíos técnicos.

Durante el desarrollo, nos encontramos con múltiples situaciones en las que ciertas funcionalidades no se comportaban como se esperaba, o directamente no estaban implementadas. Esto nos llevó a participar activamente en los repositorios oficiales, creando distintos *issues* y colaborando en la identificación y resolución de errores (ver Anexo 19.4).

Una de las principales limitaciones fue la falta de soporte para **WebRTC-Direct** en la implementación de LibP2P en JavaScript, una funcionalidad crítica para poder conectar nodos web con nodos independientes sin requerir servidores intermediarios. Esta característica se encontraba en pleno desarrollo durante el transcurso del proyecto, lo que implicó retrasos y momentos de incertidumbre respecto a la viabilidad técnica de la solución.

Afortunadamente, gracias al activo mantenimiento y la receptividad de los desarrolladores de las distintas herramientas involucradas, fue posible colaborar en la resolución de varios de estos problemas, seguir de cerca los avances en nuevas funcionalidades, y continuar con el desarrollo hasta alcanzar una infraestructura funcional y operativa.

11.2.3. Aplicabilidad al caso de uso

Uno de los desafíos para desarrollar y mantener una aplicación en IPFS es asegurar que el contenido se mantenga disponible. Nuestras implementaciones cuentan con la posibilidad de que los usuarios puedan colaborar. Estos nodos colaboradores, que se diferencian del resto de los usuarios, contribuyen a este objetivo: fijan el contenido para que otros nodos puedan recuperarlo a través de ellos. Esto representa un esfuerzo adicional que no existe en el ecosistema blockchain. Además, debido a la ausencia de *sharding* del contenido, fijar todo el contenido puede implicar una gran cantidad de almacenamiento dedicado exclusivamente a este fin.

Sitio web estático El despliegue de un sitio web estático es un caso ideal para IPFS. Esto se debe a que, al ser un sistema de archivos distribuido, IPFS permite almacenar páginas web de forma eficiente y acceder a ellas fácilmente mediante identificadores únicos. Estas características lo hacen especialmente adecuado para este uso. Otra ventaja importante es el tamaño relativamente pequeño y estable del sitio web. Como no crece de manera significativa, los colaboradores pueden fijar los archivos sin preocuparse por un aumento considerable en el espacio requerido.

Repositorio de conocimiento En este caso, la alta disponibilidad y el acceso confiable a los artículos son fundamentales. Por lo tanto, IPFS no resulta tan óptimo, ya que para lograr una alta disponibilidad sería necesario contar con muchos nodos colaboradores conectados de forma constante.

Mensajero en tiempo real En una aplicación de mensajería, aunque el historial de chats es útil, no es esencial para su funcionamiento principal. Esto reduce la necesidad de contar con múltiples nodos colaboradores. Por otro lado, OrbitDB se desempeña bien en contextos de tiempo real una vez establecida la conexión entre nodos, lo que lo convierte en una opción adecuada para este caso de uso.

11.2.4. Performance

La performance en IPFS se ve afectada por algunas variables. Entre ellas se encuentran:

- Uso de cache por parte de nodos de IPFS o gateways cuando se recupera un archivo.
- Cercanía al nodo correspondiente a la hora de publicar un CID en la Distributed Hash Table.
- Configuración y capacidades del nodo que tiene el contenido que se requiere.
- Cantidad de nodos alojando el contenido que se requiere.

Se puede minimizar el efecto de estas variables en la medida final sin distorsionar las métricas obtenidas. Más adelante se verán las maneras en las que se puede lidiar con estas variables.

Sitio Web Estático La métrica que se decidió medir es la del **tiempo que tarda un nodo en desplegar un sitio web o contenido**. Para ello, se creó un clúster con un único nodo y un repositorio Git con contenido de distinta forma.

Obtención de las métricas En este caso, el proceso de despliegue se contiene dentro del contenedor `watcher`. Para medir el tiempo real que transcurre en cada paso del despliegue, se utilizó el comando de GNU `time` para cada paso, y el resultado es sumado para obtener el tiempo total que tardó desplegar el contenido.

Variables consideradas Las métricas obtenidas se lograron ajustando dos variables: el tamaño total del contenido, y la cantidad de archivos del mismo. Los archivos en sí fueron generados repitiendo un `UUID` hasta alcanzar el número de bytes deseados. Se utilizó este tipo de identificador para asegurar de que ningún archivo permanezca en alguna caché de la red o en la DHT, y a su vez no se repitan los CID entre archivos de la misma prueba.

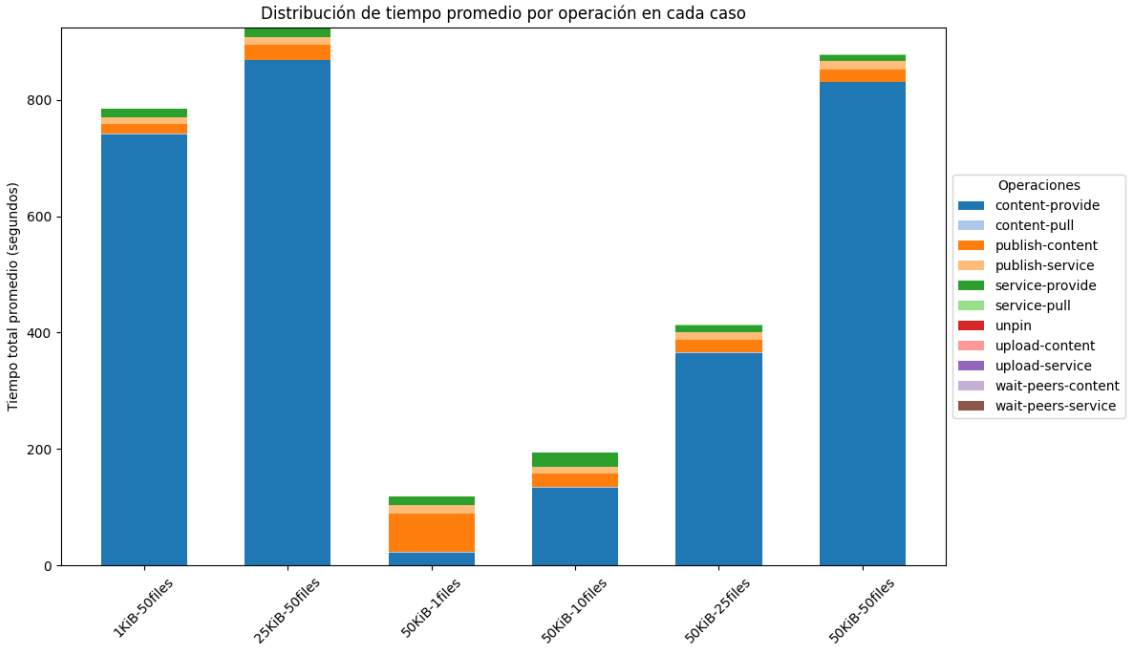

Figura 27: Distribución del tiempo promedio para desplegar el contenido para cada caso

De este gráfico podemos concluir que la operación más significante en términos de tiempo en el despliegue es la *providing* del contenido. Esto es, la publicación de todos los CIDs en la DHT. Luego, le siguen el *providing* del `service.json`, y las operaciones para actualizar el valor al que apunta el nombre de IPNS, tanto del contenido como del `service.json`. Las demás operaciones se pueden despreciar.

	Max	Mean	Min	Std	Median
1KiB-50files	23.65 s	17.66 s	7.70 s	6.39 s	21.71 s
25KiB-50files	67.29 s	25.92 s	8.96 s	15.24 s	22.95 s
50KiB-1files	78.75 s	66.09 s	61.48 s	5.83 s	62.93 s
50KiB-10files	30.30 s	23.03 s	8.93 s	3.68 s	23.37 s
50KiB-25files	23.79 s	22.65 s	21.94 s	0.46 s	22.57 s
50KiB-50files	23.36 s	20.95 s	8.17 s	4.27 s	22.43 s

Tabla 2: Estadísticas para `publish-content`

	Max	Mean	Min	Std	Median
1KiB-50files	17.76 s	10.30 s	7.54 s	3.39 s	8.79 s
25KiB-50files	23.03 s	12.76 s	7.31 s	6.37 s	8.74 s
50KiB-1files	23.08 s	15.37 s	7.52 s	6.71 s	15.25 s
50KiB-10files	23.36 s	11.86 s	6.86 s	5.79 s	8.61 s
50KiB-25files	23.02 s	12.17 s	7.43 s	5.80 s	8.97 s
50KiB-50files	22.85 s	13.91 s	7.28 s	6.43 s	11.32 s

Tabla 3: Estadísticas para `publish-service`

	Max	Mean	Min	Std	Median
1KiB-50files	851.26 s	740.32 s	682.64 s	41.45 s	734.15 s
25KiB-50files	918.36 s	868.11 s	815.43 s	26.57 s	872.25 s
50KiB-1files	42.09 s	21.84 s	10.79 s	7.49 s	20.14 s
50KiB-10files	165.63 s	133.95 s	95.91 s	18.23 s	135.24 s
50KiB-25files	412.30 s	364.53 s	306.15 s	26.53 s	366.01 s
50KiB-50files	921.54 s	830.06 s	744.83 s	41.12 s	826.65 s

Tabla 4: Estadísticas para `content-provide`

	Max	Mean	Min	Std	Median
1KiB-50files	27.62 s	14.83 s	8.84 s	6.12 s	11.41 s
25KiB-50files	34.34 s	15.67 s	10.99 s	6.64 s	11.66 s
50KiB-1files	25.71 s	13.63 s	6.49 s	5.26 s	11.34 s
50KiB-10files	35.17 s	23.21 s	11.47 s	6.95 s	25.91 s
50KiB-25files	20.78 s	12.42 s	9.78 s	3.47 s	11.26 s
50KiB-50files	20.27 s	11.31 s	6.82 s	3.17 s	10.97 s

Tabla 5: Estadísticas para `service-provide`

Teniendo en cuenta esto, se puede observar cuál es la causa de la variación del tiempo de *providing* ajustando las dos variables mencionadas.

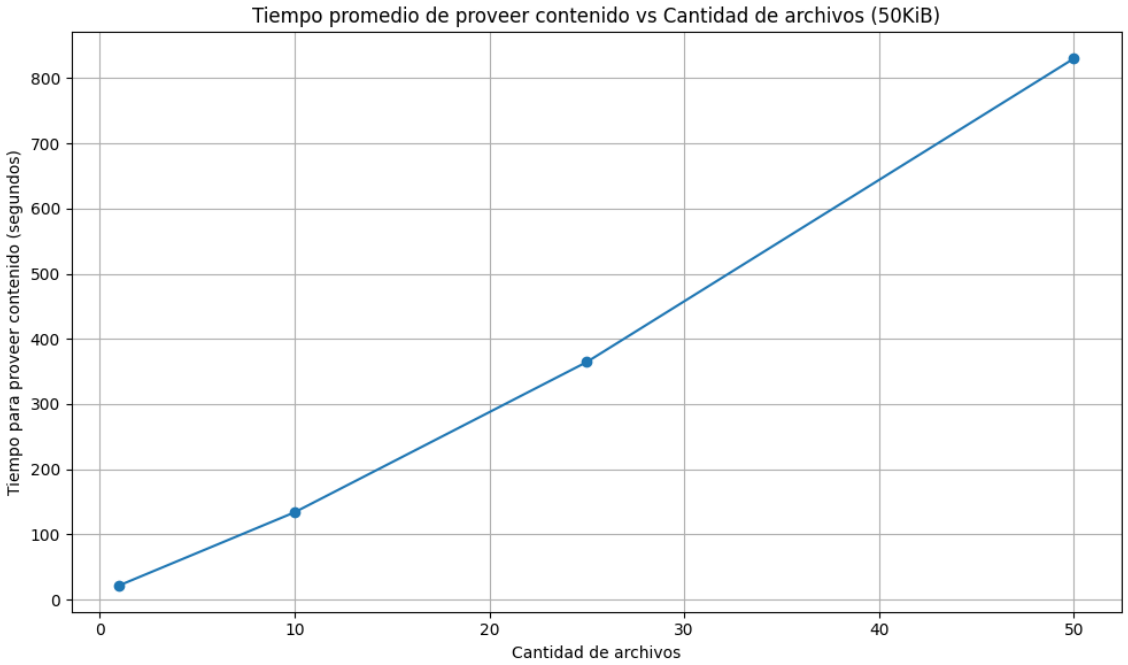

Figura 28: Tiempo promedio por cantidad de archivos

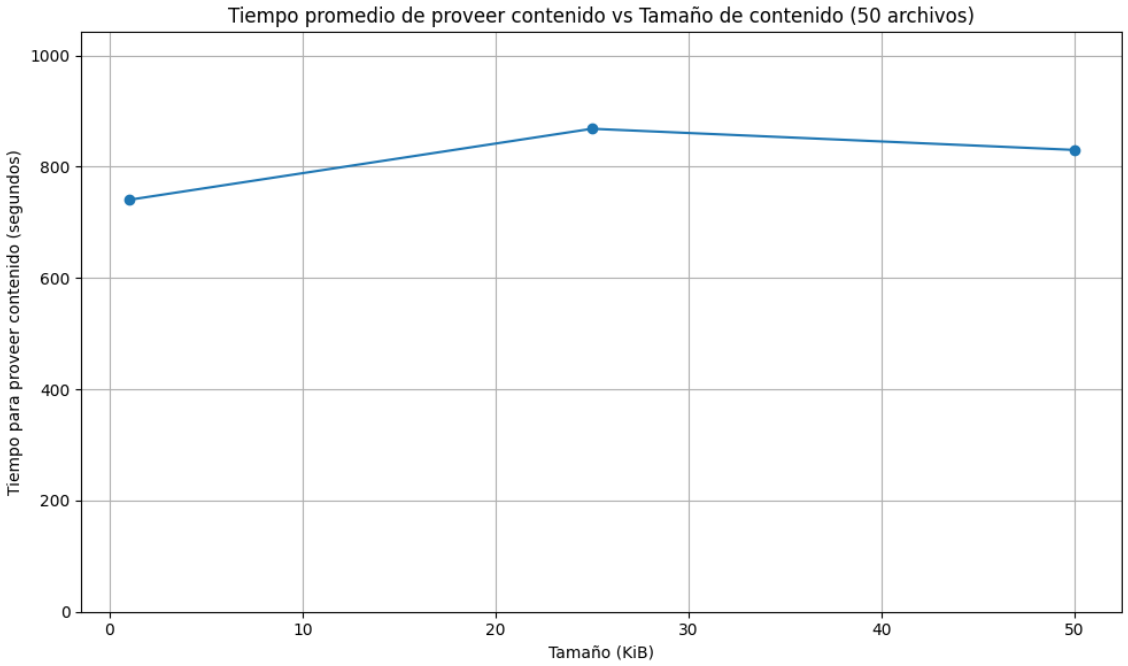

Figura 29: Tiempo promedio por tamaño del contenido

La cantidad de archivos que tiene el contenido a publicar es la variable que modifica drásticamente el tiempo que tarda un nodo confiable para desplegarlo, como se ve en las figuras. Esto se debe a que la publicación del CID del directorio no es suficiente para que otro nodo pueda obtener el contenido de ese directorio. En cambio, se requiere que se publique todos los archivos y directorios que componen el contenido. Esto también se demuestra con el tiempo constante para publicar el archivo `service.json`, ya que en todos los casos sigue siendo un sólo archivo.

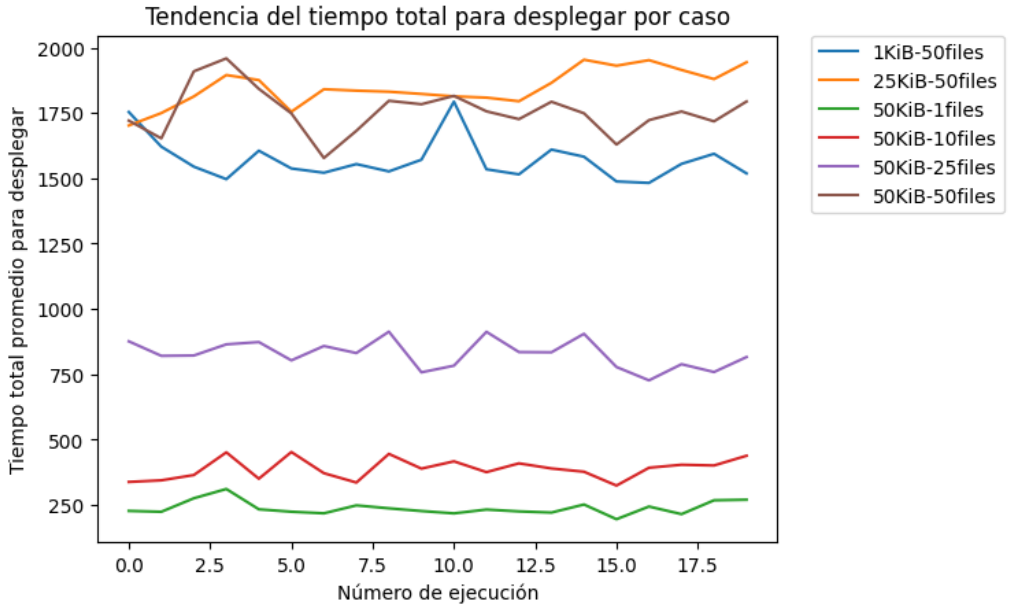

Figura 30: Tendencia del tiempo total promedio para desplegar el contenido en ejecuciones consecutivas.

Por último, se ve como la tendencia en ejecuciones consecutivas no afecta el tiempo que se tarda

en desplegar.

Repositorio de conocimiento Las métricas tanto de Astrawiki como Astrachat se ven muy relacionadas, debido a que el grueso del trabajo relacionado a IPFS se realiza con su biblioteca en común, AstraDB. Por ejemplo, el tiempo que tarda un nodo en obtener una nueva versión de un articulo es el similar al que tarda un nodo en recibir un mensaje. Por ello, se prefirió complementar las métricas obtenidas para acentuar el uso específico de AstraDB de cada caso de uso, y evitar métricas repetidas.

Creación de artículos Esta métrica fue obtenida tomando medidas del tiempo que tarda un nodo en crear el artículo. Para el tamaño del artículo, se obtuvo métricas para tamaños del contenido en incrementos de 1.000 bytes, hasta 50.000 bytes.

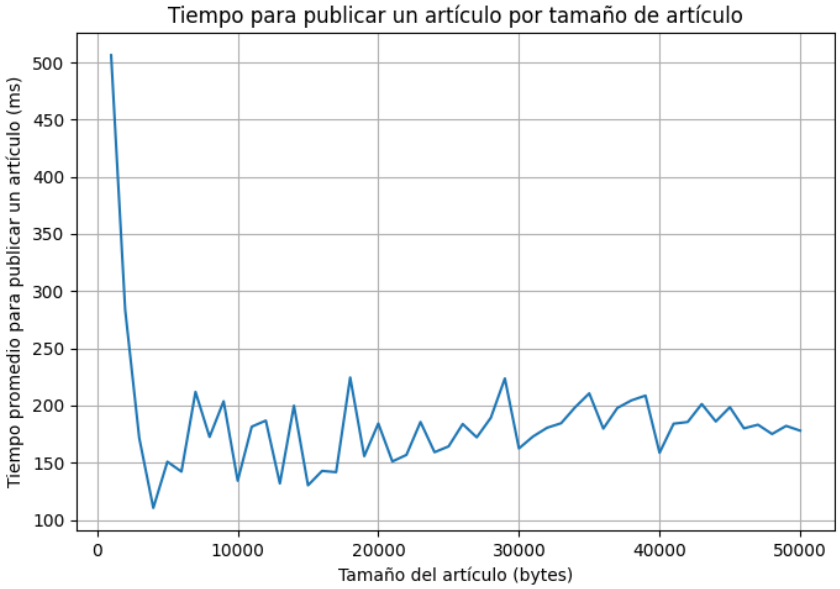

Figura 31: Tiempo para crear artículos

Del gráfico se desprende, en primer lugar, que la primera creación suele requerir más tiempo. Además, se observa que el tamaño de cada artículo no influye de manera significativa en el tiempo de publicación, debido a que el proceso de creación y sincronización de las bases de datos conllevan un tiempo mayor.

Obtención de artículos En este caso, se tomaron en cuenta los tamaños de artículos descritos previamente. Para cada uno, se realizó diez ejecuciones de treinta artículos creados, y se las promedió para lograr un tiempo estimado por acción consecutiva.

	max	**mean**	**min**	**std**	**median**
short	5086 ms	1579 ms	704.2 ms	972.8 ms	1337 ms
medium	5726 ms	1378 ms	572.3 ms	922.8 ms	1151 ms
large	6821 ms	1532 ms	709.6 ms	1101 ms	1244 ms

Tabla 6: Tiempo en obtener artículos

La presencia de picos altos se debe a la pérdida de conexión esporádica de los nodos involucrados. Sin embargo, se puede observar que la obtención de artículos no varía significativamente teniendo en cuenta los distintos tamaños. La conexión de las bases de datos individuales terminan siendo el factor que más afecta al tiempo total.

Mensajero en tiempo real Las siguientes métricas se obtuvieron con un nodo colaborador y un nodo común. Se levantaron ambos nodos en la misma máquina para poder coordinar ambos nodos y facilitar la obtención de métricas.

Se asume una conexión existente entre el nodo colaborador y el nodo común. Para agilizar la obtención de métricas la conexión directamente mediante las *multiaddresses* de los nodos, aunque en un caso real se debería encontrar utilizando la DHT. Una vez establecida la conexión el uso es el mismo, por lo que estas métricas no dependen del tipo de mecanismo utilizado para el establecimiento de la conexión.

Tiempo en enviar un mensaje A continuación se muestran los resultados de medir el tiempo que le lleva a un nodo enviar un mensaje a un chat. Para este caso, se asume que el nodo ya está conectado, situación a la que debería llegar un usuario de todos modos antes de enviar un mensaje.

	Max	Mean	Min	Std	Median
short	24230 ms	185.1 ms	30.25 ms	765.3 ms	161.4 ms
medium	21870 ms	175.8 ms	26.44 ms	690.1 ms	153.4 ms
large	121000 ms	265.5 ms	20.59 ms	3822 ms	145.5 ms

Tabla 7: Tiempo en enviar un mensaje

Algo a notar son los máximos para cada medición. Estos se deben a una pérdida en la conexión entre nodos que puede ocurrir en el transcurso de una sesión. Sin embargo, son pocos los casos dado el tamaño de la muestra. Por otro lado, el tamaño del mensaje no afecta de forma apreciable el tiempo de envío.

Tiempo en obtener mensajes El gráfico presenta cómo varía el tiempo de respuesta, medido en milisegundos, a medida que crece la cantidad de mensajes en un chat, desde 0 hasta 1000 mensajes en el chat. Se consideran tres tipos de mensajes, y se observa un incremento progresivo en el tiempo necesario para obtenerlos. Esta métrica se obtuvo midiendo el tiempo requerido para recuperar todos los mensajes en chats con diferentes tamaño de contenido. Además, se eliminaron los *outliers* por pérdida de conexión para poder visualizar mejor la tendencia en general.

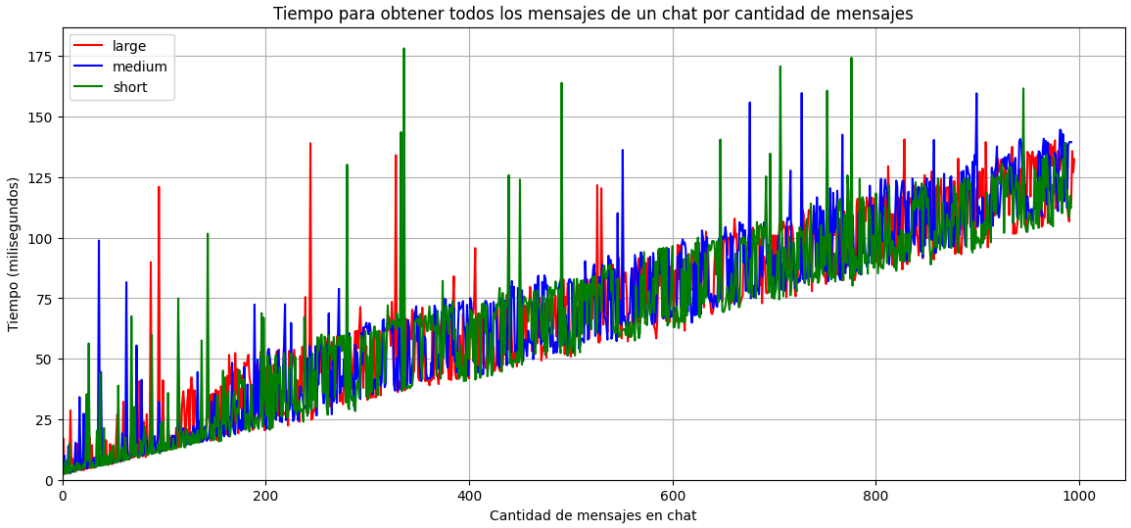

Figura 32: Tiempo para obtener mensajes de un chat según el tamaño del chat

	Max	Mean	Min	Std	Median
short	5665 ms	79.45 ms	2.340 ms	248.8 ms	69.19 ms
medium	8471 ms	85.47 ms	2.718 ms	359.4 ms	70.14 ms
large	3905 ms	76.96 ms	2.642 ms	161.2 ms	69.15 ms

Tabla 8: Tiempo en obtener mensajes

Tiempo entre enviar y recibir un mensaje Esta métrica calcula el tiempo que tarda el nodo común en recibir un mensaje enviado por el nodo colaborador. Para ello se utiliza el `EventEmitter` integrado en Astrachat, logrando una medición exacta del trayecto de un mensaje para una conexión ya establecida. Al igual que en los demás casos, existen *outliers* provenientes de pérdidas de conexión, los cuáles se pueden observar en los máximos obtenidos. Sin embargo, la desviación estándar y el valor promedio y mediano indican que estos no son frecuentes.

	Max	Mean	Min	Std	Median
short	22790 ms	298.1 ms	47.85 ms	1341 ms	198.8 ms
medium	18050 ms	227.0 ms	40.89 ms	800.4 ms	195.0 ms
large	20680 ms	285.8 ms	64.47 ms	1264 ms	196.1 ms

Tabla 9: Tiempo entre enviar y recibir un mensaje

11.2.5. Sistema utilizado

Para realizar las métricas de IPFS, se utilizó el siguiente sistema:

- **CPU:** Ryzen 5 1600
- **RAM:** 8GB
- **Almacenamiento:** SSD

11.3. Blockchain

11.3.1. Costos

Swarm Al deployar el sitio web es necesario contar con *postage stamps* que son la manera de pagar por el uso del almacenamiento en Swarm. Cada actualización que se realice al sitio requiere de *postage stamps* y, además, estos tienen fecha de vencimiento por lo que es necesario volver a pagar frecuentemente. Dichos *postage stamps* se pagan en la criptomoneda BZZ que fluctúa de valor con respecto al dólar estadounidense. La obtención del sitio web no requiere de costo alguno, por lo que desde el punto de vista de un usuario lector de la aplicación no sería necesario pagar.

El valor de los *stamps* también depende del tamaño del contenido a ser desplegado y el tiempo que se quiere mantener el contenido en la red, esto lo manejan con variables llamadas *depth* y *amount* que uno puede variar para obtener el almacenamiento y tiempo requerido. Realizando un despliegue desde un nodo de Swarm que apunta a la *testnet* Sepolia, un *postage stamp* con 1GB de almacenamiento por 2 días (*depth* 20 y *amount* 10000000000) tiene un costo de 1.0486 BZZ que equivalen a 0.1767 USD (al día 11 de junio de 2025).

Ethereum Se utiliza la moneda ETH para pagar por cada transacción, esto incluye tanto el despliegue de cada *smart contract* como también cada modificación al estado de los mismos. Por lo tanto, el usuario final de la aplicación termina pagando por la creación y edición de cada artículo en el repositorio de conocimiento, y por cada mensaje enviado en el mensajero en tiempo real. Por otro lado, para las operaciones de lectura no se tiene que pagar nada. A este monto que tiene que pagar el usuario se lo llama gas y varía con respecto a las operaciones que realiza el método que se

ejecuta al llamar a un *smart contract* y, también, con respecto a la congestión de la red, es decir a mayor congestión mayor será el costo.

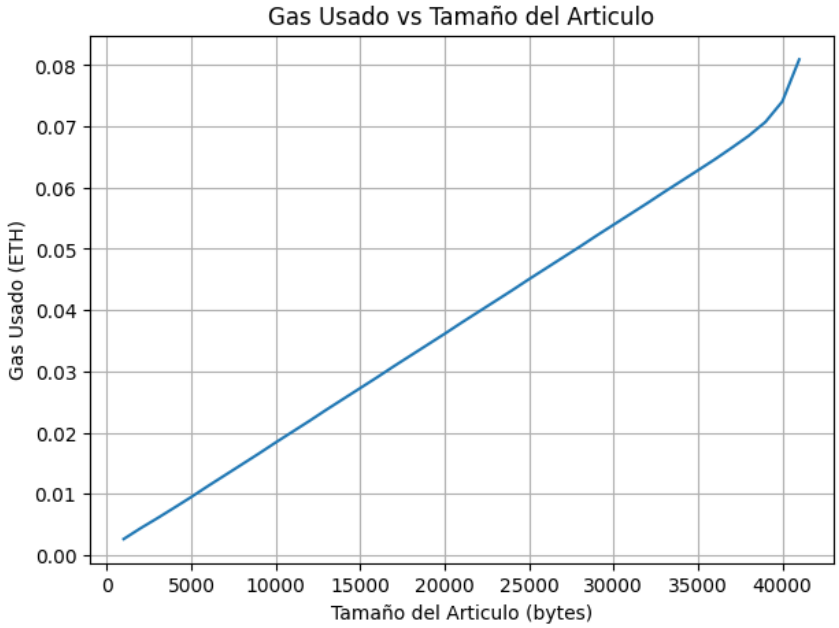

Figura 33: Gas usado al crear artículos con tamaño de bytes creciente

11.3.2. Experiencia de desarrollo

Swarm La experiencia de desarrollo en Swarm podría ser mejor. Cuenta con una comunidad pequeña pero activa. Si bien la documentación en varios aspectos puede resultar escasa, tienen medios por los cuales se pueden consultar dudas al equipo de desarrollo y son respondidas con brevedad.

Existen algunas herramientas que facilitan el desarrollo, por ejemplo, *swarm-cli* [84] la cual permite la interacción con un nodo de Swarm para realizar despliegues o consultas sobre su estado. También el equipo de Swarm provee una Github Action que permite la posibilidad de automatizar el despliegue generando un *pipeline* que utilice dicha herramienta.

En cuanto a un ambiente de pruebas o *staging*, si bien no existe un *gateway* público que interactúe con la *testnet*, es posible levantar uno propio que sí lo haga apuntando a la *testnet* de Sepolia usando la herramienta *gateway-proxy* [38].

Ethereum La experiencia de desarrollo en Ethereum resulta positiva por su gran comunidad y herramientas que provee. Por ejemplo, con la librería `web3.js` se puede interactuar con un nodo de Ethereum y realizar un despliegue de la aplicación. Además, con las herramientas de Hardhat se puede levantar una red de prueba que facilita el desarrollo local. Por otro lado, cuenta con una extensa documentación que detalla el funcionamiento de su blockchain.

11.3.3. Aplicabilidad al caso de uso

Swarm Resulta más conveniente para sitios web o recursos estáticos, al igual que IPFS. Por otro lado, al ser una tecnología de almacenamiento no es posible la ejecución de código.

A diferencia de IPFS, Swarm cuenta con incentivos incluidos (por medio de la moneda BZZ), esto significa que para deployar contenido en la red es necesario pagar. Al hacerlo te asegura que el mismo va a estar disponible durante el tiempo equivalente al costo pagado, es decir, que no es necesario pinear los archivos puesto que se encuentran en la red con un TTL.

Ethereum Su punto fuerte es la ejecución de código, por lo cual es útil para funcionar como *backend* de aplicaciones web. Como hemos visto, por el costo de almacenamiento de los *smart contracts*, no es recomendable para recursos como imágenes, videos o incluso *strings* de texto muy largos como lo realizado en el repositorio de conocimiento.

Los eventos pueden resultar útil para la interacción en tiempo real requerida en el mensajero, pero lo positivo de esto queda opacado por el hecho de necesitar pagar por cada interacción, en el caso del mensajero por cada mensaje enviado. Esto se puede volver costoso rápidamente, además de tedioso al momento de utilizar la aplicación. Se pueden explorar alternativas para reducir esta fricción, como por ejemplo, que el contrato tenga un balance de *tokens* para ser gastados, lo cual haría que el usuario no tenga que confirmar cada transacción de mensaje enviado si no que directamente el contrato lo extrae de su balance; entre otras posibilidades.

11.3.4. Performance

Para el caso del sitio web estático las métricas obtenidas se realizaron mediante un nodo local apuntando a la *testnet* Sepolia. Los valores son una aproximación a lo que sería en la *mainnet*. Para el caso de la *mainnet* habría que tener en cuenta que el tiempo de red puede ser mayor.

Las métricas para el repositorio de conocimiento y para el mensajero en tiempo real se obtuvieron levantando una instancia local de Hardhat [39]. Los tiempos obtenidos corresponden al que le toma a la transacción ejecutarse en una máquina local. En un caso real, existe un tiempo mayor de red y de interacción del usuario con la *wallet* para confirmar la transacción.

Sitio Web Estático

Despliegue de contenido Al realizar el despliegue de sitios con distintas cantidades de archivos y tamaños se obtuvieron los resultados de la Tabla 10. Vemos que la cantidad de archivos no tiene un efecto en el tiempo de publicación del contenido mientras el tamaño total del sitio se mantenga igual.

	Max	**Mean**	**Min**	**Std**	**Median**
1KiB-50files	13.93 s	8.95 s	5.24 s	1.74 s	8.83 s
25KiB-50files	9.92 s	7.99 s	5.81 s	1.19 s	8.38 s
50KiB-1files	10.39 s	7.65 s	5.75 s	1.13 s	7.79 s
50KiB-10files	22.65 s	8.84 s	5.49 s	3.73 s	7.59 s
50KiB-25files	10.41 s	8.20 s	5.66 s	1.18 s	8.24 s
50KiB-50files	14.59 s	8.62 s	5.36 s	1.80 s	8.52 s

Tabla 10: Estadísticas para desplegar contenido en Swarm

Repositorio de conocimiento

Tiempo en obtener un artículo En la siguiente tabla podemos ver los resultados de obtener 1000 muestras de la obtención de un mismo artículo en los distintos tamaños. No se nota diferencia significativa entre los tamaños *short* y *medium*, pero sí se puede ver un salto en la latencia al obtener un artículo de tamaño *large*.

	Max	**Mean**	**Min**	**Std**	**Median**
short	42.03 ms	11.26 ms	6.44 ms	3.28 ms	10.51 ms
medium	33.35 ms	15.30 ms	9.11 ms	3.47 ms	14.66 ms
large	135.32 ms	47.46 ms	31.05 ms	7.82 ms	46.34 ms

Tabla 11: Tiempo en obtener un artículo

Tiempo de creación de artículos Se registró el tiempo de creación de un artículo aumentando el tamaño en bytes del mismo. La Figura 34 muestra el resultado de tomar el promedio de 5 muestras por tamaño de artículo, aumentando el mismo de a 1000 bytes por iteración.

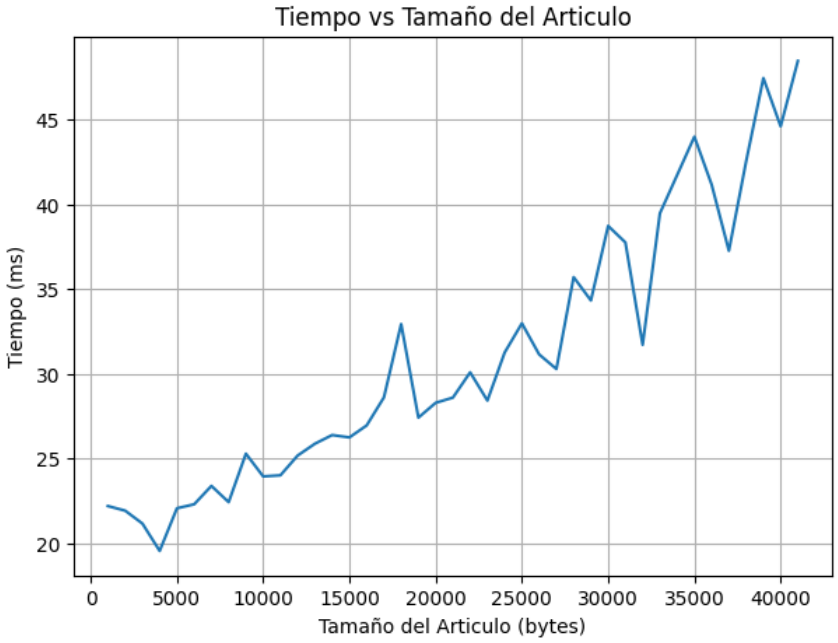

Figura 34: Tiempo para crear artículos con tamaño de bytes creciente

Mensajero en tiempo real

Las siguientes métricas se obtuvieron levantando una instancia local de Hardhat [39] y ejecutando 850 muestras y 3 veces cada una.

Tiempo en enviar un mensaje La primer métrica tomada corresponde al tiempo que tarda en enviarse un mensaje.

	Max	Mean	Min	Std	Median
short	10.93 ms	6.12 ms	5.09 ms	0.74 ms	6.01 ms
medium	10.34 ms	6.26 ms	5.24 ms	0.60 ms	6.20 ms
large	10.68 ms	6.44 ms	5.36 ms	0.70 ms	6.37 ms

Tabla 12: Tiempo en enviar un mensaje

Se puede observar que los tiempos entre los tamaños de mensajes no varían demasiado.

Gas usado para enviar un mensaje La siguiente tabla muestra el valor (en ETH y USD) de enviar el mismo mensaje corto. El precio tomado para la conversión de ETH a dólar es el de la fecha del 5 de junio de 2025 a las 19:00 hs de $2439.17 de la página Coinmarketcap [21].

		Max	Mean	Min	Std	Median
short	**ETH**	0.00042043	0.00034257	0.00034221	0.00000331	0.00034221
	USD	1.03	0.84	0.83	0.01	0.83
medium	**ETH**	0.00060668	0.00051325	0.00051274	0.00000434	0.00051274
	USD	1.48	1.25	1.25	0.01	1.25
large	**ETH**	0.00085774	0.00074335	0.00074264	0.00000579	0.00074264
	USD	2.09	1.81	1.81	0.01	1.81

Tabla 13: Precio y gas usado para enviar un mensaje

Según los resultados obtenidos se puede decir que para enviar un mensaje cuesta más de 1 dólar.

Tiempo en obtener mensajes Para esta métrica se tomó el tiempo en obtener todos los mensajes para un chat el cual iba teniendo cada vez más mensajes (desde 0 hasta 850 mensajes). El gráfico muestra para los 3 tipos de mensajes, cómo el tiempo (en milisegundos) se va incrementando a medida que hay más mensajes en el chat.

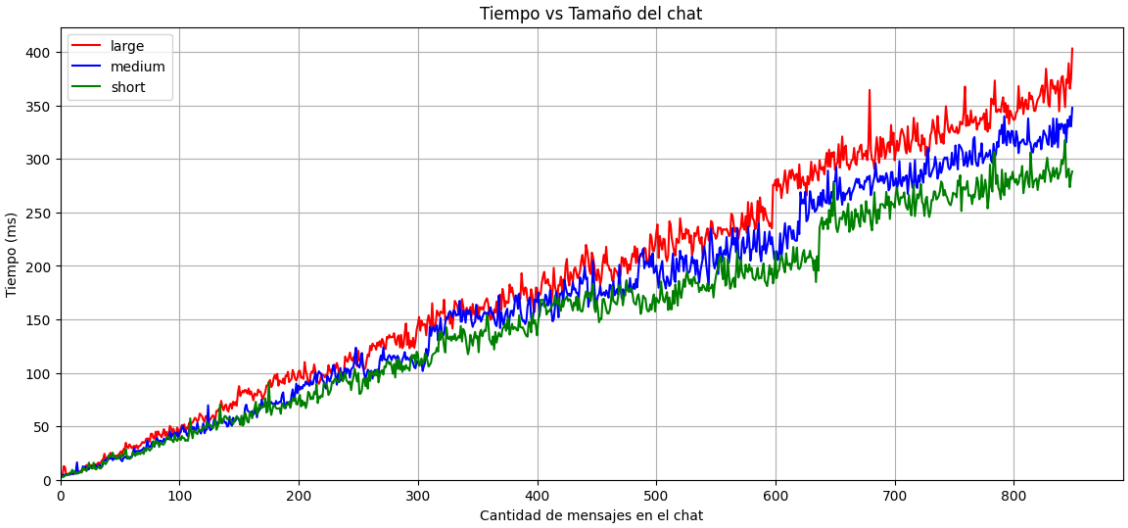

Figura 35: Tiempo para obtener mensajes de un chat según el tamaño del chat

Tiempo entre enviar y recibir un mensaje (mismo usuario) En esta métrica se midió el tiempo que tarda en un mensaje desde que es enviado a ser recibido por el canal de escucha de nuevos mensajes para un mismo usuario.

	Max	Mean	Min	Std	Median
short	38.12 ms	5.42 ms	4.26 ms	1.62 ms	4.66 ms
medium	28.97 ms	5.76 ms	4.31 ms	1.74 ms	5.05 ms
large	24.66 ms	5.76 ms	4.46 ms	0.94 ms	5.6 ms

Tabla 14: Tiempo en enviar y recibir un mensaje (mismo usuario)

Tiempo entre enviar y recibir un mensaje (distintos usuarios) Esta métrica calcula el tiempo que tarda un segundo usuario en recibir un mensaje enviado por un primer usuario.

	Max	Mean	Min	Std	Median
short	13.19 ms	5.61 ms	4.59 ms	0.97 ms	5.53 ms
medium	10.92 ms	5.84 ms	4.53 ms	1.28 ms	5.06 ms
large	13.95 ms	5.96 ms	4.65 ms	0.72 ms	5.89 ms

Tabla 15: Tiempo en enviar y recibir un mensaje (distintos usuarios)

Si comparamos las tablas de tiempos de enviar y recibir un mensaje (14 y 15) vemos que los tiempos son parecidos. Si bien los máximos de la primer tabla son mayores que los de la segunda y los datos están más dispersos, para los demás valores de la segunda tabla estos tienden a ser ligeramente mayores. En este caso particular, si se hiciera una prueba entre 2 usuarios de distintas partes del mundo el tiempo de conexión seguramente pesaría más.

11.3.5. Sistema utilizado

Para realizar las métricas del sitio web estático y el repositorio de conocimiento en Blockchain, se utilizó el siguiente sistema:

- **CPU:** Ryzen 5 1600
- **RAM:** 16GB
- **Almacenamiento:** SSD

En cuanto al mensajero en tiempo real, las especificaciones del sistema en el cual se realizaron las métricas son las siguientes:

- **CPU:** Ryzen 7 5700X
- **RAM:** 32GB
- **Almacenamiento:** HDD

11.4. Resumen

Resumiendo, y a modo de comparación entre ecosistemas, podemos concluir con lo siguiente:

	IPFS	Blockchain	Hyphanet	Freenet
Costos	Bajos o nulos	Escala con el uso de la aplicación	Bajos o nulos	Bajos o nulos
Experiencia de desarrollo	Neutra	Muy positiva	Muy negativa	N/A
Aplicabilidad para SWI	Alta	Alta	Baja	N/A
Aplicabilidad para RC	Alta	Alta	N/A	N/A
Aplicabilidad para MTR	Alta	Media	N/A	N/A
Performance	Aceptable	Aceptable	N/A	N/A

Tabla 16: Comparación entre los ecosistemas de IPFS y Blockchain

De los ecosistemas analizados blockchain es el único que conlleva costos monetarios a los usuarios finales de los casos de uso, así como también para el despliegue de los mismos. En el resto de ecosistemas no existen costos por parte del usuario final, sólo el costo de almacenamiento y energía eléctrica para aquellos que hosteen un nodo.

En cuanto a experiencia de desarrollo, blockchain es el que más herramientas y documentación provee en contraste con Hyphanet donde la documentación es escasa. Para Freenet, como no se llegó a desarrollar ninguna aplicación, no nos es posible dar un veredicto. Por la parte de IPFS, surgieron problemas al utilizar la implementación en Javascript/TypeScript y con librerías como OrbitDB sin embargo, en el resto la experiencia fue positiva.

Sobre la aplicabilidad de los casos de uso, tanto IPFS como blockchain pueden ser utilizados para los tres casos. En particular blockchain creemos que, para aplicaciones de tiempo real como chats, no es el ecosistema indicado por la mala usabilidad al tener que aceptar cada transacción. Para Hyphanet sólo pudimos probar el caso del Sitio Web Estático y, si bien en la práctica es sencillo, no es accesible a la web abierta.

Por último, la performance en ambos ecosistemas probados ha sido dentro de los parámetros esperables que una aplicación en Web2 podría tener.

Hay que tener en cuenta que las métricas realizadas para el ecosistema blockchain fueron hechas en un ambiente local, lo cual puede variar dependiendo las especificaciones de la máquina en las que fueron ejecutadas esto también implica que los tiempos de red pueden ser mayores en la realidad. Para una comparación más exacta sería necesario realizar las métricas para el ecosistema blockchain en un ambiente productivo que, por el posible alto costo monetario asociado, no fue posible realizar en este trabajo. Sin embargo, puesto que los costos de *gas* son en gran medida afectados por los *opcodes* del código del *smart contract* resultante, podemos decir que es una buena aproximación.

De la misma forma, el entorno en el que se realizaron las métricas de IPFS consistían de dos nodos ejecutándose en una misma máquina. Además, ya que la conexión no era el foco de las pruebas, se conectaron los nodos de forma directa, sin hacer uso de la DHT. Esto no representa un cambio mayor en los resultados, pero si afecta a los valores obtenidos luego de una pérdida de la conexión, ya que esta hubiera tomado más tiempo. Sin embargo, para los casos frecuentes en los que la conexión se mantuvo, las medidas deberían ser representativas.

12. Cronograma de las actividades realizadas

Inicialmente realizamos un cronograma tentativo en forma de diagrama de Gantt de la totalidad del trabajo, incluyendo el desarrollo de cada caso de uso, el despliegue en cada ecosistema y su documentación asociada como se muestra en la Figura 36. Sin embargo, a medida que fuimos desarrollando el trabajo nos encontramos con que dividir el trabajo por ecosistemas era más eficiente que dividirlo por casos de usos como en nuestro cronograma estimado.

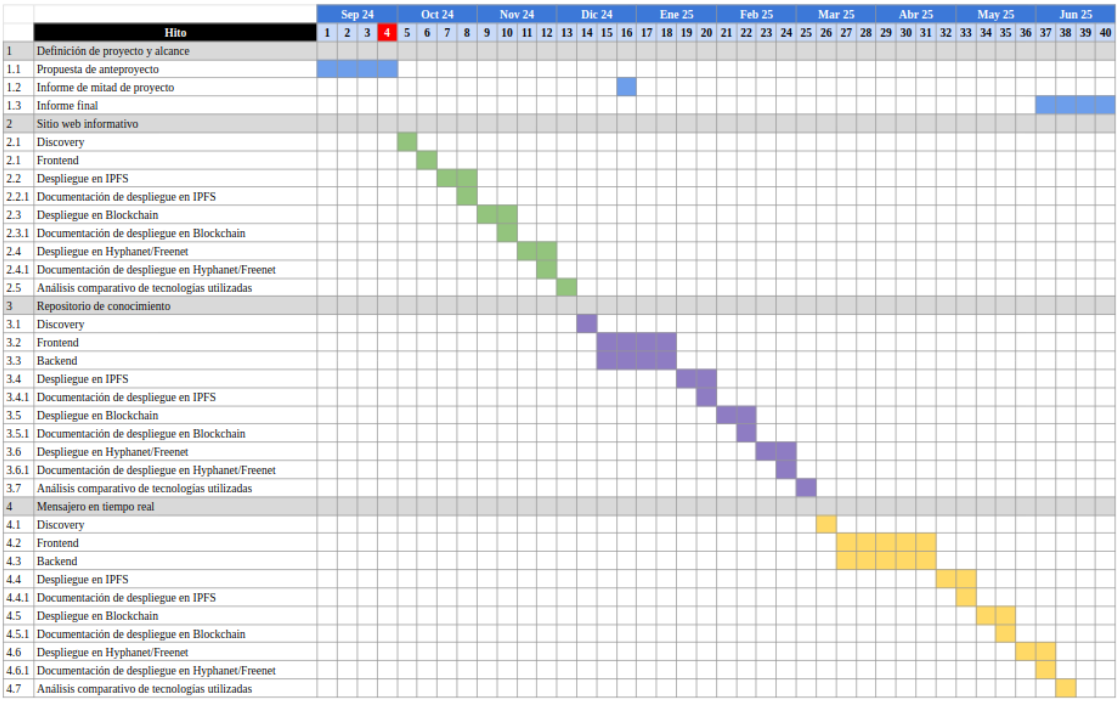

Figura 36: Cronograma tentativo

Por lo tanto, lo que terminó sucediendo es lo que se muestra en la Figura 37. La división se hizo por ecosistema, esto incluye la investigación, el desarrollo y la documentación de cada caso de uso para el ecosistema en cuestión. Hacerlo de esta manera nos permitió paralelizar los esfuerzos y evitamos superposiciones durante la investigación y desarrollo. Esta separación podría haber generado silos de conocimiento por ecosistema dentro del equipo, es por esto que las reuniones semanales de puesta en común, junto con la constante comunicación por chat y sesiones de *pair-programming*, fueron de vital importancia para mitigar dicha separación y que todo el equipo esté al tanto de los conocimientos adquiridos de cada ecosistema.

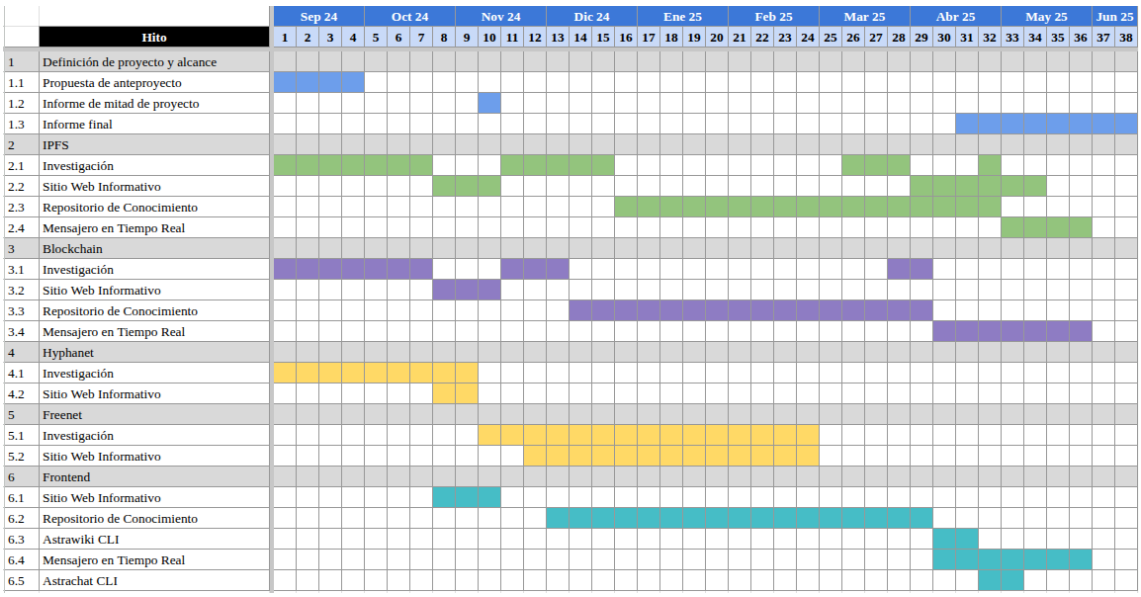

Figura 37: Cronograma real

En el cronograma real se puede ver que el desarrollo del repositorio de conocimiento en IPFS llevó más tiempo a comparación del mensajero en tiempo real. Esto fue debido a que, a medida que se fue desarrollando el repositorio de conocimiento, se fueron realizando abstracciones (como la separación en *packages* y AstraDB) que permitieron que el desarrollo del mensajero en tiempo real sea más sencillo y lleve menos tiempo. Algo similar sucedió para blockchain, donde notamos que los contratos para ambos casos de uso resultaron ser bastantes similares.

Además, antes del desarrollo de cada caso de uso dedicamos un tiempo a la investigación en cada ecosistema, esto incluyó: análisis de herramientas existentes, confección de *User Story Mappings* y lectura de documentación de cada ecosistema.

Por último, podemos notar el momento justo en el que dejamos de realizar el desarrollo de Hyphanet y Freenet, que fue en donde entró en juego nuestro plan de contingencia.

13. Riesgos materializados

Descripción	**Causa**	**Plan de Respuesta**	**Umbral**	**Plan de Contingencia**
Incapacidad de utilizar Hyphanet como ecosistema	Documentación desactualizada y/o API poco documentada	Tomar como referencia otras aplicaciones	Las aplicaciones de referencia no siguen un estándar	Descartar Hyphanet y reemplazar por Freenet
Incapacidad de utilizar Freenet como ecosistema	Ecosistema inestable debido al desarrollo activo	Esperar por versión estable	Versión estable para febrero de 2025	Dar de baja Freenet y agregar métricas de performance para los otros ecosistemas

Tabla 17: Riesgos materializados

Cambio de Hyphanet a Freenet Aproximadamente un mes luego del inicio del proyecto resolvimos cambiar el tercer ecosistema elegido (Hyphanet) por su versión más moderna (Freenet). Esto fue debido a que encontramos que la documentación era escasa, los programas realizados para el ecosistema eran unos pocos y cada uno tenía una forma distinta de implementar ciertas partes. La API tampoco provee facilidades a la hora de gestionar archivos, manejo de comunicaciones, entre otras cosas que consideramos necesarias para los casos de uso.

Freenet en desarrollo Un riesgo que teníamos en cuenta eran las modificaciones que podría sufrir Freenet al estar aún en desarrollo. Esto fue de la mano con que la documentación publicada no está actualizada a la última versión.

Baja de Freenet como ecosistema Dada la promesa del equipo de Freenet de lanzar una versión estable en el corto plazo –pero que ya llevaba más de un año en ese estado– decidimos poner como límite el mes de febrero de 2025. Llegada la fecha, no hubo ningún anuncio de la versión estable (y al momento de redactar este informe tampoco lo hay) por lo que decidimos descartar el ecosistema y, en cambio, agregar métricas de performance a los otros ecosistemas.

14. Lecciones aprendidas

14.1. Tecnologías emergentes

Trabajar con tecnologías emergentes resulta un desafío al encontrarse en desarrollo constante y frecuente. Esto quiere decir que la documentación es escasa, nula o se encuentra desactualizada. Las bibliotecas pueden cambiar su comportamiento entre versiones, incorporar nuevas funcionalidades inestables o incluso presentar errores no detectados previamente.

Estas dificultades técnicas nos llevaron a interactuar directamente con los proyectos involucrados, revisar el código fuente y abrir *issues* en los repositorios oficiales, detallando los problemas, proponiendo ejemplos mínimos reproducibles y, en algunos casos, sugiriendo soluciones. Esta experiencia, aunque desafiante, resultó sumamente formativa.

Nos permitió entender de forma práctica cómo funciona la colaboración en proyectos de software libre. Aprendimos a comunicarnos de manera clara y efectiva con los mantenedores, a estructurar nuestros aportes para que fueran útiles a otros usuarios y a seguir los canales y convenciones propios del ecosistema *open source*. En varios casos, nuestras contribuciones derivaron en mejoras directas en las bibliotecas que utilizamos.

14.2. Creación de paquetes

Durante parte del desarrollo del repositorio de conocimiento, trabajamos la implementación de cada ecosistema dentro del *front-end*. Esta decisión nos pareció razonable luego de elegir usar un mismo *front-end* para ambos ecosistemas y abstraernos en una interfaz en común. Sin embargo, una vez avanzado en ambos ecosistemas, las limitaciones de esta elección se hicieron aparentes.

Por un lado, nos restringía para realizar métricas y tests manuales fácilmente, ya que cada prueba involucraba levantar un servidor local con la página y recompilar el mismo con cada cambio, además de no tener fácil acceso para pruebas automáticas.

Por otro lado, limitaba el potencial de las herramientas que estábamos desarrollando. El resultado final es mucho más versátil ya que puede acoplarse a cualquier *front-end* como se demostró con los *front-ends* alternativos, ventaja que se alinea con la filosofía de aplicaciones descentralizadas y comunitarias.

En retrospectiva desarrollar los casos de uso de forma modular desde el comienzo hubiera agilizado el proceso, e implicaría posiblemente el uso de pruebas en una etapa más temprana.

14.3. Otras lecciones

- Paralelizar las tareas por ecosistema nos ayudó a tener un referente para cada uno y no abarcar la misma información al investigar.
- Realizar reuniones semanales nos sirvió para compartir el conocimiento adquirido de los distintos ecosistemas y ponernos al tanto de en qué estaba trabajando cada uno.
- Nos abrió al mundo P2P y web descentralizada, que es una manera distinta de pensar y desarrollar aplicaciones.

15. Impactos sociales y ambientales

A continuación se realizará un análisis del posible impacto que pueda tener el uso de estas herramientas, manteniendo la distinción entre ecosistemas.

15.1. Aplicaciones en IPFS

15.1.1. Moderación y Censura

Nuestras aplicaciones desarrolladas con IPFS carecen de moderación para los artículos y chats. Esto es debido a la falta de roles, necesarios para una moderación efectiva dentro de cada comunidad. Como se vio previamente, las bases de datos que representan artículos o chats son *append-only*, y por lo tanto no hay una manera de eliminar un contenido en particular sin eliminar la base de datos.

Tampoco es sencillo eliminar la base de datos. Tanto en Astrawiki como en Astrachat, cada chat/artículo es único, y por lo tanto no se puede simplemente sobrescribir el contenido con otra base de datos. Por otro lado, la base de datos siempre estará disponible mientras otras personas la alojen.

La manera más factible de eliminar un artículo o chat por completo es a nivel de *front-end*, o sea, manualmente eliminando resultados de las búsquedas y no permitiendo la carga de esos artículos o chats específicos. Esto se puede realizar siempre y cuando se tenga acceso al código fuente del *front-end* utilizado, y se tenga permisos para modificarlo de manera que los usuarios obtengan la nueva versión al usarlo.

Sin embargo, nada impide que otro *front-end* distinto maneje esos mismos datos públicos de forma diferente, según sus propias políticas o criterios.

Todas las ventajas y limitaciones de IPFS abarcan nuestras implementaciones. En el caso en que un individuo o entidad -como un gobierno, o un ISP- quiera censurar estos recursos, no basta con prohibir un dominio en particular, ya que IPFS es abierto y los CIDs y nombres de IPNS seguirán disponibles. Existen maneras de censurar por CID que explotan componentes de IPFS como la Distributed Hash Table, pero pueden ser mitigados. [83] Tampoco es útil inhabilitar un *gateway* en específico, ya que los usuarios pueden optar por usar otra *gateway*, o directamente acceder mediante su propio nodo de IPFS.

Por otro lado, los nodos de IPFS no son anónimos. El `PeerID` de cada nodo es permanente y se mantiene cada vez que se inicia. A la vez, se puede relacionar el `PeerID` de un nodo con una *multiaddress*, y por lo tanto, la IP pública del nodo [74]. No obstante, incluso obteniendo la IP de uno de los nodos que aloja el contenido, este suele estar disponible en múltiples nodos. Esto incluye a cualquier usuario que utilice la aplicación y *gateways* que la recuperen, ya que por la naturaleza de IPFS, el contenido se mantendrá en la caché del nodo por un tiempo.

La contra-cara de estas dificultades para censurar un contenido alojado en IPFS es la facilidad para confirmar que un contenido ya no está disponible. Un nodo puede saber con certeza que un contenido no está disponible simplemente intentando recuperarlo, lo cuál es difícil de lograr con servidores HTTP [32].

Además, un estudio realizado en Septiembre de 2023 [13] afirma que gran parte de los nodos presentes en la red de IPFS provienen de servicios de *cloud*, los cuáles son propietarios y fáciles de manipular para censurar o restringir el acceso a un contenido en específico. Entre otras estadísticas, se resalta que un 79,6 % de los nodos en la DHT provienen de *datacenters*, y un 80 % de los nodos que están detrás de una red con NAT, obtienen contenido mediante *relays* alojados en servicios de *cloud*.

Los servicios desarrollados con IPFS como infraestructura son muy resilientes a posibles ataques. A su vez, la dificultad para eliminar contenido no deseado presenta un posible problema, ya que se puede utilizar para alojar contenido malicioso, ilegal, o simplemente no acorde a una comunidad que aloje una wiki o chat.

Acceso a información en regiones con censura o poca conexión Un beneficio claro del uso de IPFS es su tolerancia a la partición de la red [32]. Debido a que el contenido puede ser replicado en múltiples nodos, no se puede completamente limitar el acceso y publicación de contenido. Por otro lado, una vez que un nodo en una red local o regional obtiene el contenido, este estará disponible en esa red mientras el nodo lo ofrezca, lo cuál es útil para lugares en donde la conexión

a Internet es limitada, ya que para obtener dicho contenido no se requiere recuperar archivos fuera de la red local.

15.1.2. Impacto ambiental

No hay análisis realizados acerca de la utilización de energía por parte de la red de IPFS en comparación con soluciones centralizadas. Sin embargo, hay varios factores que diferencian el uso de energía de una solución centralizada con una realizada con IPFS como infraestructura.

Recuperación de contenido En un escenario óptimo, el contenido requerido está disponible en un nodo cercano. Por ejemplo, si otro usuario en un área especifico accede a un contenido, el resto de los nodos en ese área podrán recuperar el contenido desde ese nodo. En cambio, un contenido alojado en un servidor HTTP siempre requiere acceso al servidor, el cuál puede estar a mayor distancia. Esto implica un uso mayor de ISPs regionales y de *tier 1*, y en general requiere infraestructura de red de mayor capacidad.

Eficiencia Dado que parte de los nodos proveedores de contenido están ubicados en la *periferia* de la red (dispositivos móviles, computadoras de hogar, etc.), los métodos de lectura y escritura de contenido no son óptimos. Los nodos pueden estar utilizando discos duros en vez de unidades de estado sólido, o en general componentes ineficientes comparados con un servidor dedicado a proveer contenido HTTP. Esto puede resultar en un mayor uso de energía.

Escalabilidad Si bien los servicios de *cloud* proveen escalabilidad para variar la cantidad de máquinas dedicadas a proveer un contenido especifico, la naturaleza de IPFS hace que esta red sea incluso más optimizada. Cuando muchos nodos requieren de un contenido, se replica por la red fácilmente y sin cargas altas para ningún nodo en particular.

Realizar un análisis detallado del impacto ambiental de IPFS es uno de los trabajos futuros propuestos.

15.2. Aplicaciones en Ethereum

Se sabe que blockchain abrió un mundo de posibilidades en cuanto a la privacidad y el impacto ambiental. A continuación desarrollaremos el impacto de las aplicaciones desarrolladas en esta tecnología.

15.2.1. Moderación y Censura

Al igual que como sucede con IPFS, en Ethereum no es posible borrar un *smart contract* de la red. Esto significa que una vez creado un artículo o un chat el mismo no puede ser borrado, haciéndolo resistente a la censura.

En cuanto a la moderación de contenido por parte de gobiernos o ISP, no es posible dado que es una red distribuida y, mientras exista un nodo que tenga la información de la red, la misma va a estar disponible para todos los usuarios.

Si se trata de moderación de contenido por parte de los propios usuarios no se puede garantizar que no surjan modificaciones en los artículos o chats. Cualquier persona con la *address* del contrato y cumpliendo con la firma de los métodos del mismo puede realizar modificaciones.

Otro impacto social a destacar es que al ser un ecosistema en donde se debe utilizar dinero para realizar modificaciones, aquellas personas con mayor poder adquisitivo tendrán una dominancia por parte de aquellas personas con menos ingreso.

15.2.2. Impacto ambiental

Desde su adopción, blockchain en general ha sido criticado por el enorme impacto ambiental y la huella de carbono al utilizar casi tanta energía eléctrica como un país entero [34]. Esto, con los años ha ido cambiando y, al día de hoy, existen blockchains *eco-friendly* [90]. En particular, la red de Ethereum pasó de utilizar *proof-of-work* a *proof-of-stake* lo cual hizo que el consumo eléctrico de los nodos disminuyera en hasta un 99 % [42]. Sin embargo, hay dudas sobre la forma de medir este impacto y compararlo con otras blockchain como Bitcoin. Además, la cantidad de nodos que se sumaron a *proof-of-stake* luego de la migración se duplicó [67].

16. Trabajos futuros

A partir del análisis realizado, se identificaron diversas líneas de trabajo que abren la posibilidad de continuar y ampliar el desarrollo presentado.

16.1. Mejoras a los casos de uso

A continuación se proponen mejoras concretas para los casos de uso implementados. Si bien no pudieron ser abordadas en esta instancia por cuestiones de tiempo, representan oportunidades claras para fortalecer y extender la solución actual.

16.1.1. AstraDB

El enfoque adoptado para AstraDB presenta ciertas limitaciones que abren la puerta a posibles mejoras futuras. A continuación, se enumeran algunas de las más relevantes:

Una primera mejora posible sería habilitar conexiones entre nodos web. Actualmente, la infraestructura sólo permite que los nodos independientes sirvan las bases de datos al resto de los usuarios. Poder establecer conexiones directas entre navegadores permitiría que los mismos usuarios web que estén utilizando la aplicación en simultáneo puedan colaborar en la distribución de datos, alineándose aún más con la filosofía descentralizada de IPFS. Esta funcionalidad requiere que los nodos web sean capaces de aceptar conexiones entrantes, lo cual solo es posible mediante el uso de un **Relay**. En el contexto de este proyecto, no se logró un funcionamiento estable del sistema de Relay, pero su incorporación una vez estabilizado sería sencilla e incrementaría significativamente la resiliencia y disponibilidad del sistema.

Por otra parte, la solución actual no priorizó aspectos de seguridad. Toda persona que conozca el nombre de una entidad puede agregar información a su base de datos correspondiente, y los colaboradores deben almacenar su contenido completo. Al no contar con un sistema de *Proof of Storage*, esto abre la posibilidad a ataques de *spam*: un usuario malicioso podría crear numerosas claves o llenar las existentes con información irrelevante, ocupando el almacenamiento de los colaboradores. Este riesgo podría mitigarse implementando un mecanismo de consenso, mediante el cual los cambios propuestos deban ser aprobados antes de ser distribuidos y aceptados. No obstante, este tipo de enfoque requiere un análisis más profundo centrado en la seguridad, que escapa a los objetivos de esta implementación inicial.

Relacionado con lo anterior, también se identifica como posible mejora el soporte para casos de uso donde se requiera control de edición más estricto. Actualmente, todas las claves son editables por cualquier usuario. Para aplicaciones como blogs personales o *microblogs*, sería deseable que solo el creador de una clave pueda modificar su contenido, mientras que el resto de los nodos actúan únicamente como replicadores. Esto podría lograrse incorporando controles de acceso más específicos, por ejemplo, estableciendo que la base de datos asociada a cada clave solo acepte modificaciones del nodo que la creó. Dado que la dirección de una base de datos incluye la clave pública de su creador, esta validación es técnicamente posible dentro del modelo de OrbitDB.

Finalmente, la solución actual no contempla la posibilidad de datos privados. Todo el contenido

se almacena públicamente en la red IPFS y no se aplica ningún mecanismo de cifrado. Por tanto, no es adecuada para aplicaciones que requieran confidencialidad. La incorporación de esquemas de cifrado a nivel de aplicación podría ser una solución viable en futuros desarrollos, permitiendo que sólo los usuarios autorizados puedan leer ciertos contenidos.

16.1.2. Astrawiki-eth

Teniendo en cuenta que es el caso de uso donde mayor volumen de datos se transfieren, por lo tanto mayor costo de gas, una mejora involucra la disminución de este costo. Se podría disminuir haciendo uso de los clones de *smart contract*s según lo propuesto en el ERC-1167 [71]. En resumen, lo que se propone es que haya un contrato como "plantilla" al cual se lo copia cada vez que, por ejemplo, se crea un artículo nuevo.

16.1.3. Astrachat-eth

Una de las principales mejoras que se pueden implementar sobre este caso de uso es respecto a la experiencia de uso para *front-ends* en navegadores web. El hecho que para cada mensaje enviado sea necesario confirmar una transacción por medio de la *wallet* hace que la aplicación pierda el factor de *real-time*. Una posible solución a este problema es que se pueda crear un balance en un fondo común (con dinero de los usuarios) de forma que cada vez que se envíe un mensaje se tome de este fondo para pagar la transacción. Esta mejora haría más fácil la integración con otros *front-ends* que no corren en navegadores web como *Astrachat-cli* aunque también queda investigar cómo conectar una *wallet* a los mismos.

Por otro lado, en base al costo por transacción (visto en la sección 11.3.4) se podría utilizar alguna red cuyos precios de gas por transacción sean menores. Una de estas redes –basada en Ethereum– es la *Ronin Network* [77] [78] que a principio de este año (2025) abrió su red para que cualquier usuario pueda implementar aplicaciones en la misma.

16.2. Mejora para clusters colaborativos

Como se analizó previamente en el contexto de IPFS, los *clusters* colaborativos representan una alternativa viable para garantizar la persistencia y disponibilidad de archivos mediante el *pinning* distribuido entre nodos participantes, permitiendo así el *hosting* comunitario.

No obstante, esta alternativa aún presenta una adopción limitada en la práctica, debido a diversas restricciones técnicas y operativas, que se detallan a continuación.

IPFS Cluster IPFS Cluster permite configurar el *replication factor*, es decir, la cantidad deseada de réplicas para cada archivo dentro del clúster. Este parámetro resulta útil para optimizar el uso del almacenamiento, evitando la duplicación innecesaria de datos en todos los nodos. Sin embargo, presenta una limitación crítica: IPFS Cluster no implementa mecanismos de verificación que aseguren que los nodos seguidores estén efectivamente realizando *pinning* del contenido asignado. Tampoco valida el espacio disponible, la integridad de los datos ni el cumplimiento efectivo del acuerdo de replicación.

Como consecuencia, un nodo malicioso puede simular estar colaborando sin almacenar realmente ningún dato. Esta falta de garantías impide aplicar con seguridad factores de replicación bajos, ya que el contenido podría volverse inaccesible si los nodos asignados no cumplen su rol. Por este motivo, la estrategia más común es replicar todo el contenido en todos los nodos, lo cual restringe severamente la escalabilidad del sistema.

Este modelo es el que se observa actualmente en los *clusters* públicos listados por el proyecto IPFS [56], donde a los colaboradores se les solicita disponer de capacidades de almacenamiento que pueden variar desde varios *gigabytes* hasta *terabytes*, limitando significativamente la participación comunitaria a gran escala.

Despliegue, seguimiento y descubrimiento Más allá de las limitaciones técnicas, existen también obstáculos operativos en cuanto a la facilidad de despliegue, monitoreo y descubrimiento de proyectos colaborativos que utilizan IPFS Cluster.

Si bien nuestra herramienta de infraestructura de despliegue facilita la publicación de contenido en *clusters* colaborativos, la incorporación de nuevos colaboradores continúa requiriendo conocimientos técnicos avanzados y la ejecución manual de comandos, lo que constituye una barrera de entrada para usuarios menos experimentados. El desarrollo de herramientas más accesibles y orientadas a la experiencia del usuario permitiría ampliar significativamente la red de participantes.

Asimismo, no existe actualmente una plataforma abierta y descentralizada que permita registrar y descubrir fácilmente proyectos comunitarios. Los únicos *clusters* públicos disponibles se encuentran listados en la página oficial de IPFS [56], donde la inclusión de nuevos proyectos depende de una gestión centralizada. Una solución que permita registrar proyectos de forma abierta, incluyendo descripciones, requisitos de colaboración y enlaces de participación, facilitaría enormemente la adopción y el involucramiento comunitario en una variedad más amplia de iniciativas distribuidas.

Conclusión Las limitaciones expuestas permiten identificar una línea de trabajo futura enfocada en la mejora de los clusters colaborativos, la cual podría abordarse en dos ejes complementarios:

- **Mejoras técnicas:** desarrollar mecanismos —ya sea mediante extensiones de IPFS Cluster o capas complementarias— que permitan validar de forma efectiva el pinneo distribuido, sin necesidad de replicación total ni dependencia de nodos completamente confiables.
- **Herramientas de participación:** diseñar una plataforma que facilite el despliegue, monitoreo y descubrimiento de aplicaciones comunitarias distribuidas, reduciendo las barreras técnicas y promoviendo la colaboración descentralizada.

16.3. Análisis del consumo de energía en la red de IPFS

Como se mencionó anteriormente, no hay análisis detallado del uso de energía en la red de IPFS. Un posible trabajo involucra calcular una huella de carbono aproximada, el consumo promedio de energía, entre otras métricas. De ello puede desprenderse un análisis comparativo entre diferentes ecosistemas como Blockchain, y *cloud hosting* tradicional.

16.4. Blockchain para aplicaciones comunitarias

En la búsqueda de una blockchain comunitaria nos encontramos con un servicio llamado *Filecoin* [59] el cual provee de incentivos monetarios para aquellos que cedan espacio en sus discos a guardar archivos de otras personas. Es abierto para que cualquier persona pueda unirse a la red *peer-to-peer* que proveen. Funciona sobre IPFS, utiliza tecnologías de Blockchain para los incentivos y dan garantía que los datos están realmente guardados en los nodos. Nuestra propuesta de trabajo futuro es que se cree una tecnología basada en Blockchain que no necesite del incentivo monetario para funcionar y que además permita la ejecución de código.

16.5. Análisis de Freenet como ecosistema

Si en un futuro Freenet finalmente sale en una versión estable, creemos que sería un buen ecosistema al cual realizar un análisis como el de este trabajo y hasta más en profundidad con otro tipo de aplicaciones distribuidas y descentralizadas.

16.6. IPFS + Ethereum

Habiendo visto las ventajas y desventajas de cada ecosistema se podría tratar de combinar ambos de forma de aprovechar los beneficios de cada uno. Por ejemplo, se podrían utilizar los

smart contract de Ethereum para guardar los CID de los artículos y chats, y utilizar IPFS para guardar el contenido de los mismos. De esta forma el costo de parte de los usuarios se traduce sólo al momento de la creación y se aprovechan la velocidad de sincronización de Ethereum con la capacidad de almacenamiento de IPFS.

17. Conclusiones

Luego de todo el análisis realizado podemos ver que cada ecosistema tiene sus ventajas y desventajas. IPFS y Swarm por su naturaleza como almacenamientos descentralizados funcionan muy bien para sitios web estáticos. En particular, IPFS tiene la gran ventaja de no requerir costos monetarios para el usuario final, si no que promueve la colaboración a través de la contribución de almacenamiento. Por otro lado, Ethereum y Swarm sí requieren un costo monetario para el usuario final pero aseguran una disponibilidad y persistencia mucho mayor. Ethereum resulta útil para aplicaciones como el repositorio de conocimiento y el mensajero en tiempo real pero se puede volver muy costoso según su implementación, y posee algunas cuestiones de usabilidad que son más evidentes en el mensajero. Además, debido a su comunidad y amplia documentación, Ethereum presenta una mejor experiencia de usuario que el resto de ecosistemas. Por su parte, Hyphanet se encuentra prácticamente abandonado y su proyecto sucesor Freenet, si bien parece prometedor, todavía está en pleno desarrollo. Concluyendo en que lo más beneficioso sea combinar lo mejor de cada ecosistema para aplicaciones como los casos de uso presentados.

El desarrollo de este trabajo nos brindó la oportunidad de explorar estas tecnologías emergentes las cuales, al estar en desarrollo constante, presentaron varios desafíos que tuvimos que afrontar como la ausencia de documentación o documentación desactualizada, cambios repentinos de la interfaz de ciertas librerías, faltas de herramientas necesarias para el desarrollo o despliegue de algún ecosistema (lo visto en IPFS), o incluso tecnologías que todavía se encuentran en desarrollo y no tienen una versión estable (como nos sucedió con Freenet). Por otra parte, logramos aplicar los conocimientos técnicos adquiridos en nuestra formación académica y nos adentramos en la comunidad *open source*.

Por último, podemos decir que adquirimos nuevas herramientas y conocimientos que previamente no poseíamos en la profundidad alcanzada en este trabajo y que, sin duda, podremos aplicar en el futuro.

18. Referencias

[1] *ACME*. (s.f.). https://letsencrypt.org/how-it-works/
[2] *Astrachat Cli*. (s.f.). https://github.com/bitxenia/astrachat-cli
[3] *Astrachat ETH*. (s.f.). https://github.com/bitxenia/astrachat-eth
[4] *Astrachat IPFS*. (s.f.). https://github.com/bitxenia/astrachat
[5] *AstraDB*. (s.f.). https://github.com/bitxenia/astradb
[6] *Astrawiki Cli*. (s.f.). https://github.com/bitxenia/astrawiki-cli
[7] *Astrawiki ETH*. (s.f.). https://github.com/bitxenia/astrawiki-eth
[8] *Astrawiki IPFS*. (s.f.). https://github.com/bitxenia/astrawiki
[9] *Astrawiki Web*. (s.f.). https://github.com/bitxenia/astrawiki-web
[10] *Astrawiki Web Trusted Peer*. (s.f.[a]). https://github.com/bitxenia/astrawiki-web-trusted-peer
[11] *Astrawiki Web Trusted Peer*. (s.f.[b]). https://github.com/bitxenia/astrawiki-collaborator
[12] *AutoTLS*. (s.f.). https://blog.libp2p.io/autotls/
[13] Balduf, L., Korczyński, M., Ascigil, O., Keizer, N. V., Pavlou, G., Scheuermann, B., & Król, M. (2023). The Cloud Strikes Back: Investigating the Decentralization of IPFS. *Proceedings of the 2023 ACM on Internet Measurement Conference*, 391-405. https://doi.org/10.1145/3618257.3624797
[14] Batt, S. (2021). Your Files for Keeps Forever with IPFS. *Opera Blog*. https://blogs.opera.com/tips-and-tricks/2021/02/opera-crypto-files-for-keeps-ipfs-unstoppable-domains/

[15] Benet, J. (2014). IPFS - Content Addressed, Versioned, P2P File System. *IPFS Project site*. https://github.com/ipfs/papers/raw/master/ipfs-cap2pfs/ipfs-p2p-file-system.pdf
[16] *Bitxenia*. (s.f.). https://github.com/bitxenia
[17] Bondy, B. (2024). IPFS Support in Brave. *Brave Blog*. https://brave.com/blog/ipfs-support/
[18] Bourke, T. (2001). *Server load balancing*. O'Reilly Media, Inc.
[19] *Censorship of Wikipedia*. (s.f.). https://en.wikipedia.org/wiki/Censorship_of_Wikipedia
[20] CLARKE, I. (1999). 'Freenet : A distributed anonymous information storage and retrieval system,' white paper of The free Network Project. *http://freenetproject.org/index.php?page=index*. https://cir.nii.ac.jp/crid/1572824499747330944
[21] *Coinmarketcap Ethereum Price*. (s.f.). https://coinmarketcap.com/currencies/ethereum/
[22] *Collaborative Clusters*. (s.f.). https://ipfscluster.io/documentation/collaborative/
[23] *Configuration Reference*. (s.f.). https://ipfscluster.io/documentation/reference/configuration
[24] *Content Addressing*. (s.f.). https://docs.ipfs.tech/concepts/glossary/#content-addressing
[25] *Daemon (computing)*. (s.f.). https://en.wikipedia.org/wiki/Daemon_(computing)
[26] *Data Differencing*. (s.f.). https://en.wikipedia.org/wiki/Data_differencing
[27] *Deploy to IPFS Action*. (s.f.). https://github.com/marketplace/actions/deploy-to-ipfs
[28] *Differences between js-libp2p in Node.js and browser*. (s.f.). https://github.com/libp2p/docs/blob/master/content/guides/getting-started/webrtc.md#differences-between-js-libp2p-in-nodejs-and-browser
[29] *Distributed Hash Tables (DHTs)*. (s.f.). https://docs.ipfs.tech/concepts/dht
[30] *Distributed wikipedia mirror*. (s.f.). https://github.com/ipfs/distributed-wikipedia-mirror/tree/cbab5fc09c622249d5433abddb0a5976e2a51875?tab=readme-ov-file#goal-2-fully-read-write-wikipedia-on-ipfs
[31] *DNSLink*. (s.f.). https://docs.ipfs.tech/concepts/dnslink/#dnslink
[32] Doan, T. V., Psaras, Y., Ott, J., & Bajpai, V. (2022). Towards Decentralised Cloud Storage with IPFS: Opportunities, Challenges, and Future Directions. https://arxiv.org/abs/2202.06315
[33] *ENS*. (s.f.). https://ens.domains/
[34] *Ethereum blockchain produced equivalent of Honduras' annual emissions before upgrade - study*. (s.f.). https://www.reuters.com/sustainability/ethereum-blockchain-produced-equivalent-honduras-annual-emissions-before-upgrade-2023-12-01/
[35] *Filebase*. (s.f.). https://filebase.com/
[36] *Fleek*. (s.f.). https://fleek.xyz/docs/platform/hosting/
[37] *Freenet*. (s.f.). https://freenet.org
[38] *Gateway Proxy*. (2025). https://github.com/ethersphere/gateway-proxy
[39] *Hardhat*. (s.f.). https://hardhat.org/
[40] *Helia: IPFS node implementation in Javascript*. (s.f.). https://github.com/ipfs/helia
[41] *Hole Punching*. (s.f.). https://docs.libp2p.io/concepts/nat/hole-punching/
[42] *How Green Is Ethereum?* (s.f.). https://www.investopedia.com/how-green-is-ethereum-2-0-6666266
[43] *HTTP API for IPFS Cluster*. (s.f.). https://ipfscluster.io/documentation/reference/api/
[44] *HTTP API for Kubo*. (s.f.). https://docs.ipfs.tech/reference/kubo/rpc/
[45] *Hyphanet*. (s.f.). https://www.hyphanet.org/index.html
[46] *Hyphanet jSite Guide*. (s.f.). https://www.hyphanet.org/pages/documentation.html
[47] *Hyphanet Plugin - Freemail*. (s.f.). https://github.com/hyphanet/plugin-Freemail
[48] *Hyphanet Plugin - Freetalk*. (s.f.). https://github.com/hyphanet/plugin-Freetalk
[49] *Hyphanet Plugin - jSite*. (s.f.). https://github.com/hyphanet/wiki/wiki/jSite
[50] *Hyphanet Plugin - Sone*. (s.f.). https://github.com/Bombe/Sone
[51] *Hyphanet Plugin - Web of Trust*. (s.f.). https://github.com/hyphanet/plugin-WebOfTrust
[52] *Hyphanet Plugin Guide*. (s.f.). https://github.com/ArneBab/freenet-plugin-bare-guide/blob/master/index.org
[53] *Ink*. (s.f.). https://github.com/vadimdemedes/ink
[54] *InterPlanetary Name System*. (s.f.). https://docs.ipfs.tech/concepts/ipns
[55] *IPFS Cluster*. (s.f.). https://ipfscluster.io/
[56] *IPFS Cluster - Collaborative Clusters*. (s.f.). https://collab.ipfscluster.io/#list-of-clusters

[57] Johnsen, J. A., Karlsen, L. E., & Birkeland, S. S. (2005). Peer-to-peer networking with BitTorrent. *Department of Telematics, NTNU*. https://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf
[58] *Kubo: IPFS node implementation in Go*. (s.f.). https://docs.ipfs.tech/install/command-line/
[59] Labs, P. (2017, 19 de julio). *Filecoin: A Decentralized Storage Network*. https://www.filecoin.io/filecoin.pdf
[60] *libp2p*. (s.f.). https://libp2p.io/
[61] *Limo*. (s.f.). https://eth.limo/
[62] *Markdown*. (s.f.). https://en.wikipedia.org/wiki/Markdown
[63] *Metamask*. (s.f.). https://metamask.io/
[64] Mittal, N. (2007). *Timestamping Messages and Events in a Distributed System using Synchronous Communication* [Tesis doctoral, University of Texas]. https://users.ece.utexas.edu/~garg/dist/dc07-timestamp.pdf
[65] *Multiaddr*. (s.f.). https://multiformats.io/multiaddr/
[66] *An O(ND) Difference Algorithm and Its Variations*. (s.f.). https://neil.fraser.name/writing/diff/myers.pdf
[67] *One Year After The Merge: Sustainability Of Ethereum's Proof-Of-Stake Is Uncertain*. (s.f.). https://www.forbes.com/sites/digital-assets/2023/10/11/one-year-after-the-merge-sustainability-of-ethereums-proof-of-stake-is-uncertain/
[68] *OrbitDB*. (s.f.). https://orbitdb.org
[69] *The Origins of Swarm*. (2015). https://blog.ethswarm.org/hive/2024/the-origins-of-swarm
[70] Petar Maymounkov, D. M. (2023). Kadmelia: A Peer-to-peer Information System Based on the XOR Metric. *Stanford Secure Computer Systems Group*. https://www.scs.stanford.edu/~dm/home/papers/kpos.pdf
[71] *Peter Murray (@yarrumretep), Nate Welch (@flygoing), Joe Messerman (@JAMesserman), "ERC-1167: Minimal Proxy Contract," Ethereum Improvement Proposals, no. 1167, June 2018. [Online serial]. Available: https://eips.ethereum.org/EIPS/eip-1167*. (s.f.). https://eips.ethereum.org/EIPS/eip-1167
[72] *Pinata*. (s.f.). https://pinata.cloud/ipfs
[73] *Pinning*. (s.f.). https://docs.ipfs.tech/concepts/glossary/#pinning
[74] *Privacy and Encryption*. (s.f.). https://docs.ipfs.tech/concepts/privacy-and-encryption
[75] *React*. (s.f.). https://react.dev/
[76] *The React Framework for the Web*. (s.f.). https://nextjs.org/
[77] *Ronin Network*. (s.f.). https://roninchain.com/
[78] *Ronin Network Whitepaper*. (s.f.). https://docs.roninchain.com/basics/white-paper
[79] *rubix-documents*. (s.f.). https://github.com/rubixvi/rubix-documents
[80] *Server Components*. (s.f.). https://react.dev/reference/rsc/server-components
[81] *Small-World Routing*. (s.f.). https://en.wikipedia.org/wiki/Small-world_routing
[82] *SQLite*. (s.f.). https://www.sqlite.org/
[83] Sridhar, S., Ascigil, O., Keizer, N., Genon, F., Pierre, S., Psaras, Y., Rivière, E., & Król, M. (2023). Content censorship in the interplanetary file system. *arXiv preprint arXiv:2307.12212*.
[84] *Swarm CLI*. (2025). https://github.com/ethersphere/swarm-cli
[85] *Swarm postage stamps*. (2025). https://docs.ethswarm.org/docs/concepts/incentives/postage-stamps
[86] Team, S. (2021). Swarm; Storage and communication infrastructure for a self-sovereign digital society. *URL https://www.ethswarm.org/swarm-whitepaper.pdf*.
[87] *A Universally Unique IDentifier (UUID) URN Namespace*. (s.f.). https://www.rfc-editor.org/rfc/rfc4122
[88] *WebRTC*. (s.f.). https://docs.libp2p.io/guides/getting-started/webrtc/
[89] *The WebSocket Protocol*. (s.f.). https://www.rfc-editor.org/rfc/rfc6455
[90] *What is a Green Blockchain? Eco-Friendly Tech*. (s.f.). https://www.casper.network/get-started/a-brief-guide-to-green-blockchain-technology
[91] *Wikipedia Statistics: Edits*. (s.f.). https://en.wikipedia.org/wiki/Wikipedia:Statistics#Edits
[92] Wood, G., et al. (2014). Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, *151*(2014), 1-32.

19. Anexos

19.1. Arquitectura del repositorio de conocimiento

La arquitectura del repositorio de conocimiento fue diseñada de manera independiente del ecosistema utilizado. Se tomó esta decisión para proveer una interfaz común, independiente de su implementación en IPFS y Ethereum. De esta manera, una aplicación puede trabajar sobre una API común e intercambiar implementaciones fácilmente. Esto es útil para la implementación del front-end del repositorio, como se verá más adelante.

En sí, la arquitectura consiste de una o varias *wikis*, dependiendo de la implementación. Cada wiki se identifica con un nombre único que representa un grupo de artículos en conjunto. Cada wiki se representa con una lista de artículos, los cuales son identificados por sus nombres —también únicos.

Un artículo se compone de un nombre único y no vacío y su contenido. Si bien nuestra implementación utiliza *markdown* [62] como formato del contenido, el tipo de contenido es arbitrario e indiferente para la wiki, cualquier tipo de archivo es admisible. Únicamente será tenido en cuenta en el momento en el que la aplicación que interactúe con el contenido lo interprete. En nuestro caso, markdown representa una manera sencilla de enriquecer un texto, lo cual lo hace apropiado para un repositorio de conocimiento comunitario.

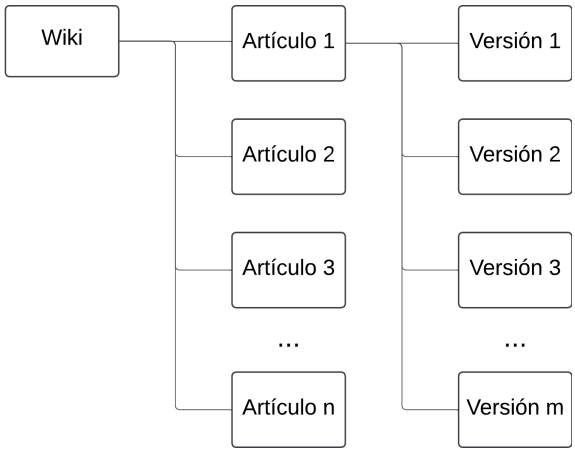

Figura 38: Arquitectura general del repositorio de conocimiento.

19.1.1. Representación de un artículo

Internamente, los artículos pueden editarse en el tiempo, y un usuario debe obtener siempre la última versión. Sin embargo, se debe poder ver versiones anteriores si así lo desea el usuario, por lo que guardar la última versión únicamente no es válido. Es por esto que se decidió representar a un artículo como un nombre, y un conjunto de versiones. Cada versión representa una modificación hecha al artículo. Un artículo nuevo tendrá una única versión, y al hacerle una edición tendrá dos.

Esta iteración de la arquitectura permite a los usuarios ver el historial de versiones, y acceder a cualquiera de ellas. Pero guardar todo un artículo por cada edición puede ser costoso en concepto de almacenamiento. Rara vez un artículo es modificado en su totalidad [91]. En cambio, las ediciones que puede sufrir un artículo se deben en su mayoría a la inclusión de contenido nuevo, o la modificación de una sección en particular. Guardar el contenido completo en casos donde los cambios son mínimos resulta ineficiente, y es particularmente negativo en aplicaciones peer-to-peer, donde el contenido se almacena en el lado del usuario.

Generación de patches Como solución para minimizar la información redundante en cada versión, se decidió usar el tipo de algoritmo de *data differencing* [26], que consiste en almacenar

únicamente los cambios hechos en una versión con respecto a su antecesora, de forma similar a *Git*. Esta lista de cambios forma un *patch*.

Un *patch* se obtiene al aplicar un algoritmo que reciba el contenido anterior, y el contenido nuevo, y devuelva un *patch* que pueda ser utilizado posteriormente para reconstruir el contenido. Si bien hay varios algoritmos que logran este objetivo, se decidió utilizar el algoritmo de *diff* de Myers [66], ya que ofrece un punto medio entre compresión, y cómputo necesario para compilar el texto. Este algoritmo es el que utiliza la biblioteca *diff-match-patch*, creada por Google, que ofrece una interfaz sencilla para lograr generar *patches* y luego compilar texto en base a ellos.

Calculando los *patches* de la nueva versión con respecto a la anterior, reducimos el espacio necesario para almacenar un artículo con todas sus versiones. Hay casos en los que un *patch* puede tener un tamaño mayor al de simplemente almacenar una copia del nuevo contenido. Sin embargo, estos casos no son frecuentes en usos comunes y decidimos despreciarlos, aunque puede ser una posible mejora para optimizar más aún el espacio utilizado.

Por otra parte, ya que cada versión no tiene toda la información, se necesitan todas las versiones anteriores para compilar el contenido resultante. Esto puede ser una desventaja en archivos con muchas ediciones, ya que requieren de mayor cómputo para compilar, específicamente `O(n)`, donde `n` es la cantidad de versiones que tiene el artículo. Para mejorar esto, una posible mejora consiste en fijar un número máximo de *patches* consecutivos que, al superarse, indica que la siguiente versión debe contener el contenido completo. De esta manera se acota la cantidad de iteraciones que puede tomar la compilación de un artículo.

La nueva iteración de la arquitectura apunta a tener una lista de versiones, que a su vez contengan el *patch* con las diferencias con la versión anterior. Pero como se verá en la siguiente sección, esta solución no es lo suficientemente resiliente.

19.1.2. Concurrencia

En las redes peer-to-peer es común que la replicación de contenido lleve un tiempo no despreciable. Esto puede generar problemas, ya que dos nodos pueden publicar contenido que entre en conflicto, de la misma manera que dos cambios pueden generar un *merge conflict* en Git. En el mejor de los casos, ambos nodos publican contenido que no entra en conflicto entre sí, y por lo tanto la compilación se puede realizar sin problemas. En el caso contrario, puede dejar el artículo en un estado inválido, imposible de compilar y por lo tanto, inaccesible.

Yendo a nuestro caso de uso, el problema planteado previamente puede entrar en conflicto de dos maneras:

1. Al crear un articulo
2. Al editar un artículo

Por lo tanto, es necesario una solución que mitigue estos problemas.

Para el caso de la creación de dos artículos con en mismo nombre, se delega en cada ecosistema la responsabilidad de detectar estos casos y actuar en consecuencia. La implementación de cada ecosistema contiene más información para tomar una decisión optimizada al respecto, la cuál se verá en secciones posteriores. En cambio, el problema de la edición de un artículo de forma concurrente puede ser resuelto a nivel arquitectura.

19.1.3. Árbol de versiones

Hasta ahora, cada versión se agrega a una lista de versiones, y se asume que cada versión se basa en la anterior. Cuando ocurre un conflicto de concurrencia entre versiones, esto no aplica por las razones dadas previamente. Por lo tanto, es necesario que cada versión tenga un ID para que las versiones posteriores puedan mantener registro de su versión padre, es decir, la versión sobre la que se basa. Todas las versiones tienen padre, excepto la versión inicial. Cuando dos versiones

se crean en base a una misma versión padre, se genera una bifurcación. Por lo tanto, las versiones pasan a formar un árbol.

En arquitecturas tradicionales, la generación de IDs suele hacerse por la misma base de datos, de manera incremental. Esto no genera problemas ya que el servidor decide en que orden tomar las versiones. En una aplicación distribuida, esto no es conveniente, ya que dos nodos pueden generar el mismo ID por la misma razón por la que pueden generar versiones conflictivas. La solución elegida fue usar `UUID`s [87], que se generan sin necesidad de centralizar la decisión, y tienen una probabilidad de colisión casi nula.

Una problemática que trae tener distintas ramas de versiones es que la compilación del contenido deja de ser trivial, ya que hay múltiples posibles últimas versiones, dependiendo de la rama que se elija. Para resolver esto es necesario definir una heurística que elija la rama principal del artículo de forma determinista.

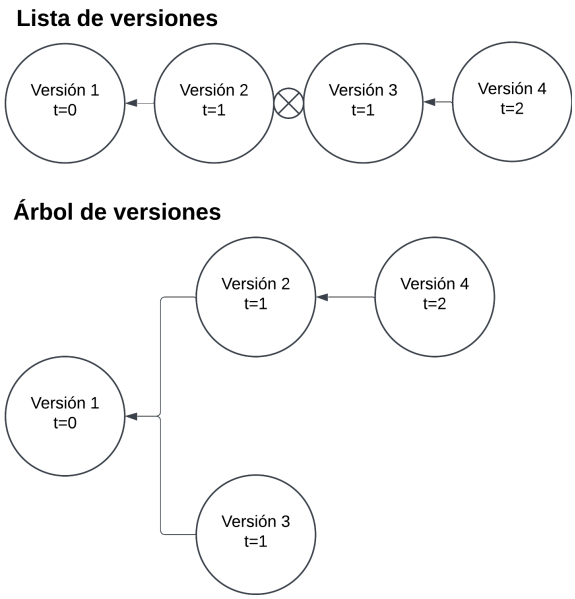

Figura 39: Ambas maneras de almacenar versiones. En este caso, `t` es un momento en el tiempo en el que todas las bases de datos están sincronizadas. En el primer caso, cuando `t=1` en ambas versiones, significa que hay un conflicto ya que ambas se basan en la misma versión padre. En el árbol, ese problema se mitiga.

Elección de rama principal Una de las decisiones tomadas respecto al árbol de versiones, es que los usuarios sólo puedan modificar la última versión "principal", es decir; un usuario no podrá editar en base a una versión arbitraria. Teniendo en cuenta esto, es fácil deducir que las ramas no principales o secundarias tendrán a lo sumo una versión en la mayoría de los casos. Esto es debido a que los demás usuarios que no hayan causado la colisión elegirán una rama principal, por lo tanto las demás ramas dejarán de recibir versiones nuevas. Elegir la rama de mayor cantidad de versiones será parte de la heurística para elegir la rama principal.

Sin embargo, esta decisión no es exhaustiva, ya que puede haber dos ramas con la misma longitud. De hecho, este caso es muy común debido a que dos usuarios que crean una nueva versión en base a la misma versión padre generan dos ramas del mismo tamaño. Para tomar una decisión que sea determinista e igual en todos los nodos, se decidió tomar la antigüedad de la última versión de cada rama como parámetro para la heurística. La rama mas antigua será considerada la rama principal. La antigüedad de una rama es decidida por una marca de tiempo o *timestamp* grabada en el mismo nodo que crea la rama, y es enviado como parte de la versión a los demás nodos. Esto trae una desventaja en cuánto a seguridad, ya que un nodo puede falsificar un *timestamp* y así tener prioridad siempre, pero es mejorable utilizando algoritmos para crear *timestamps* en un sistema distribuido [64].

```
type VersionID = string;

type Version = {
  id: VersionID;
  date: string;
  patch: Patch;
  parent: VersionID | null;
};
```

Figura 40: Propiedades de una `Versión`

Una vez elegida la versión principal, se considera como rama principal a todas las versiones que entren en la cadena de padres, empezando por la versión principal hasta llegar a la versión raíz. De esta forma, sabemos que la rama principal siempre se podrá compilar, generando una red mucho más resiliente a cambios concurrentes. Por último, las versiones que no sean parte de la rama principal también podrán visualizarse. Se tomó esta decisión para permitir que los usuarios que hayan subido una versión por fuera de la rama principal puedan ver el contenido que editaron y, en todo caso, editar la nueva versión principal con el mismo.

19.1.4. Arquitectura del mensajero en tiempo real

La arquitectura para un mensajero es mucho más sencilla que la del repositorio de conocimiento. Consta de un grupo de chats, que a su vez contienen todos los mensajes. Cada mensaje contiene el texto enviado, el usuario que lo envió, un *timestamp* del momento de su envío, entre otras cosas.

Aliases En las implementaciones que se verán posteriormente, se determina que la manera de identificar a un usuario será mediante un ID similar a un *hash* en ambos casos. Esto es común para sistemas descentralizados, pero vuelve difícil diferenciar los usuarios en un chat. Para solucionar esto, se decidió agregar un alias que identifique al usuario, que no tenga que ser único y pueda ser cambiado. Su implementación varía dependiendo del ecosistema, pero en ambos es parte del objeto `ChatMessage` que almacena cada chat.

Respuestas Se decidió identificar cada mensaje mediante un identificador, de manera tal que se pueda indicar el "padre"de un mensaje, o sea, el mensaje al que se le está respondiendo.

```
export type ChatMessage = {
  id: string;
  parentId: string;
  sender: string;
  senderAlias: string;
  message: string;
  timestamp: number;
};
```

Figura 41: Propiedades de un `ChatMessage`

19.2. Elección de herramientas en la arquitectura de despliegue de IPFS

A lo largo del diseño de la arquitectura de despliegue en IPFS, se tomaron decisiones técnicas que influyeron directamente en la descentralización, persistencia, y accesibilidad del contenido. En esta sección, se presentan las principales alternativas consideradas y el razonamiento detrás de las herramientas finalmente seleccionadas.

19.2.1. Persistencia

IPFS Cluster fue la tecnología utilizada para manejar la persistencia de archivos en la red de IPFS. Sin embargo, existen alternativas a esta herramienta que tienen sus ventajas a pesar de no ser idóneas para una aplicación comunitaria.

Servicios de *pinning* Una manera sencilla de garantizar la disponibilidad del contenido en IPFS es mediante servicios de *pinning*, como Fleek [36], Filebase [35], o Pinata [72]. Estos servicios alojan el contenido en varios nodos propios, eliminando la necesidad de contar con infraestructura local.

Sin embargo, este enfoque entra en conflicto con la filosofía de descentralización. Primero, suelen tener planes gratuitos con funcionalidad limitada, lo cual restringe su escalabilidad comunitaria. Segundo, se introduce una dependencia directa de un tercero: si el servicio deja de fijar los archivos, estos pueden quedar inaccesibles o incluso perderse. Esto representa una forma de centralización delegada, más cercana al modelo de *cloud hosting* tradicional que a una verdadera aplicación descentralizada.

Filecoin Filecoin [59] es una red de almacenamiento incentivado que permite contratar espacio a cambio de FIL, su criptomoneda nativa. Los proveedores de almacenamiento deben demostrar regularmente que conservan los datos mediante pruebas criptográficas.

A diferencia de los servicios de *pinning*, los nodos de Filecoin no están bajo una misma entidad organizativa, lo que lo vuelve más descentralizado. No obstante, su funcionamiento implica un costo económico continuo, lo cual lo hace poco viable para una solución comunitaria sin fines de lucro. Además, la complejidad técnica y operativa es mayor comparada con alternativas como IPFS Cluster.

19.2.2. Acceso y mutabilidad

Estas herramientas facilitan el acceso del contenido, y la posibilidad de cambiar el contenido sin modificar la forma de acceder. En la solución se eligió como herramienta para el acceso a ENS, y para la mutabilidad, IPNS. Sin embargo, DNSLink fue una alternativa posible para ambos.

DNSLink [31] permite asociar un nombre de dominio tradicional con un CID o nombre de IPNS mediante registros DNS TXT. Es rápido, compatible con navegadores, y ofrece nombres legibles para humanos. Además, puede apuntar a direcciones IPNS, facilitando la mutabilidad del contenido.

Sin embargo, DNS se basa en una infraestructura centralizada. Autoridades como ICANN controlan la raíz del sistema, lo que introduce vulnerabilidades ante censura o intervención por parte de registradores e ISPs. Este nivel de dependencia es incompatible con el objetivo de una aplicación verdaderamente comunitaria y resistente a la censura.

19.3. Selección del protocolo de transporte para conexiones web en IPFS

Durante el desarrollo de los casos de uso sobre el ecosistema IPFS, fue necesario tomar decisiones clave respecto al manejo de conexiones entre nodos, especialmente al tratar con aplicaciones accesibles desde navegadores web. Este anexo describe cómo se llegó a la elección final del protocolo de transporte utilizado en la arquitectura para permitir conexiones desde un nodo web (en navegador) hacia un nodo independiente.

Al comienzo del proyecto, la implementación JavaScript de LibP2P ofrecía dos alternativas principales para habilitar esta conectividad.

WebSockets La primera alternativa fue el uso de **WebSockets** [89], un protocolo que permite establecer una conexión persistente y bidireccional a través de una conexión HTTP inicial. No obstante, para que funcione en navegadores modernos, requiere el uso de certificados TLS válidos, lo cual implica contar con un dominio propio y un certificado emitido por una autoridad certificadora.

Dado que la infraestructura buscaba permitir que cualquier usuario pudiera levantar un nodo de forma autónoma, sin depender de servidores centralizados o dominios externos, esta alternativa fue descartada.

Hole Punching (Relay) La segunda opción fue el uso de la técnica conocida como **Hole Punching** [41], mediante el uso de nodos *Relay*. Estos permiten que nodos detrás de NAT o *firewalls* (como los navegadores) puedan conectarse a otros nodos a través de un intermediario.

Aunque este enfoque era prometedor, su implementación en JavaScript aún era inestable al momento del desarrollo del proyecto, lo que imposibilitó su adopción efectiva.

AutoTLS Frente a estas limitaciones, surgió durante el desarrollo el soporte experimental para una nueva alternativa: **AutoTLS** [12]. Este mecanismo permite obtener automáticamente un certificado TLS asociado al ID del nodo, utilizando el protocolo ACME [1]. LibP2P provee además un servicio DNS público que resuelve los dominios requeridos para todos los nodos participantes, lo que permite establecer conexiones WebSocket seguras sin intervención del usuario.

Esta solución fue adoptada inicialmente y permitió avanzar con el desarrollo y las pruebas. No obstante, AutoTLS presentó dos inconvenientes críticos. Por un lado, su funcionamiento resultó inestable, ya que se trata de una funcionalidad aún experimental. Por otro, depende de un servicio centralizado para la emisión de certificados, lo cual contradice el principio de descentralización del proyecto. En caso de que dicho servicio quedara fuera de línea, los nodos no podrían conectarse entre sí.

WebRTC-Direct Finalmente, se incorporó soporte para una nueva alternativa que ya estaba disponible en otras implementaciones de LibP2P: **WebRTC**, y en particular su variante **WebRTC-Direct**, diseñada para conectar directamente navegadores con nodos independientes.

WebRTC es un conjunto de estándares abiertos y APIs que permiten a los navegadores establecer conexiones seguras *peer-to-peer* sin necesidad de certificados TLS. LibP2P adapta este protocolo para permitir exactamente lo que se requería: conexiones directas desde el navegador sin intermediarios ni servidores externos.

Hasta ese momento, incluso los propios desarrolladores de LibP2P recomendaban utilizar implementaciones en otros lenguajes, como Go, ya que la versión JavaScript no ofrecía aún una solución madura para este tipo de conectividad [28]. Sin embargo, cambiar de lenguaje no era una opción viable, ya que el ecosistema del proyecto —incluyendo OrbitDB— está fuertemente vinculado al entorno JavaScript, y además se requería compatibilidad con entornos web.

Con la integración de WebRTC-Direct fue posible reemplazar completamente a AutoTLS y establecer conexiones estables y descentralizadas desde navegadores hacia nodos independientes, cumpliendo tanto con los requisitos técnicos como con los principios filosóficos del proyecto.

19.4. Listado de issues reportados en herramientas del ecosistema de IPFS

Durante el desarrollo en el ecosistema de IPFS se identificaron diversos problemas en las bibliotecas utilizadas —incluyendo IPFS, libp2p y OrbitDB—. Estos fueron reportados mediante *issues* en los repositorios oficiales, acompañados de descripciones detalladas, ejemplos mínimos reproducibles y, en algunos casos, propuestas de mejora o soluciones parciales. A continuación se enumeran los casos reportados:

1. **OrbitDB**
   - `fix: Update relay example to use default reservation ttl setting`
     https://github.com/orbitdb/orbitdb/pull/1206
   - `Peer Discovery in OrbitDB Using Only the Database Address`
     https://github.com/orbitdb/orbitdb/issues/1207
   - `Error building OrbitDB app with Next.js`
     https://github.com/orbitdb/orbitdb/issues/1214
   - `OrbitDB crashes with UnexpectedEOFError when a peer joins and syncs`
     https://github.com/orbitdb/orbitdb/issues/1226
2. **libp2p-js**
   - `Unable to establish an outbound relay connection`
     https://github.com/libp2p/js-libp2p/issues/2897
   - `AutoTLS IPv4 Multiaddr(s) Do Not Upgrade`
     https://github.com/libp2p/js-libp2p/issues/2929
   - `Certhash missing from appendAnnounce webrtc-direct multiaddrs`
     https://github.com/libp2p/js-libp2p/issues/3080

19.5. Repositorios

Todos los repositorios de las aplicaciones y paquetes desarrollados se encuentra bajo la organización **Bitxenia** [16]:

- IPFS
  - Astradb [5]
  - Astrawiki [8]
  - Astrachat [4]
  - Astrawiki-web-trusted-peer [10]
  - Astrawiki-collaborator [11]
- ETH
  - Astrawiki-eth [7]
  - Astrachat-eth [3]
- Front-end
  - Astrawiki-web [9]
  - Astrawiki-cli [6]
  - Astrachat-cli [2]