Informe de Trabajo Profesional

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

Integrantes

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

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

Joaquín Matías Velazquez
105980
jvelazquez@fi.uba.ar

Joaquín Prada
105978
jprada@fi.uba.ar

Tutor
Ariel Scarpinelli
ascarpinelli@fi.uba.ar

Índice

1. Resumen **4**

2. Palabras Clave **4**

3. Abstract **4**

4. Keywords **4**

5. Introducción **4**

6. Estado del Arte **5**
6.1. Introducción a arquitecturas de red 5
6.2. Diferencias y ventajas de cada arquitectura 5
6.3. Ambientes y herramientas 7
6.3.1. IPFS 7
6.3.2. Blockchain 8
6.3.3. Alternativas 8

7. Problema detectado y/o faltante **8**
7.1. Costos 8
7.2. Interrupciones del servicio 9
7.3. Zonas de censura 9

8. Solución implementada **9**
8.1. Casos de uso 9
8.1.1. Sitio web informativo 9
8.1.2. Repositorio de conocimiento 10
8.1.3. Mensajero en tiempo real 14
8.2. Proceso de descubrimiento 15
8.3. IPFS 15
8.3.1. Infraestructura de despliegue 15
8.3.2. Infraestructura de aplicación 23
8.4. Blockchain 27
8.4.1. Swarm 27
8.4.2. Ethereum 27
8.5. Front-end 29

9. Metodología **32**

10. Experimentación y/o validación **32**
10.1. Costos 32
10.1.1. IPFS 32
10.1.2. Blockchain 32

10.2. Experiencia de desarrollo 33
10.2.1. IPFS 33
10.2.2. Blockchain 33
10.3. Viabilidad 33
10.3.1. IPFS 33
10.3.2. Blockchain 33
10.4. Performance 34
10.4.1. IPFS 34
10.4.2. Blockchain 34
10.5. Resumen 36

11. Cronograma **36**

12. Riesgos materializados **37**

13. Lecciones aprendidas **38**

14. Impactos sociales y ambientales **38**

15. Trabajos futuros **38**

16. Conclusiones **38**
16.1. Conclusión del análisis 38
16.2. Conclusión general 38

17. Referencias **38**

18. Anexos **40**
18.1. Herramientas de LibP2P utilizadas 40

1. Resumen

En el siguiente trabajo se analizan distintos ecosistemas y tecnologías que se pueden utilizar para el despliegue de aplicaciones web comunitarias de manera distribuida y descentralizada.

Mediante tres casos de uso que ilustran diferentes características: un sitio web informativo, un repositorio de conocimiento y un mensajero en tiempo real, se comparan ventajas y desventajas del despliegue de cada uno de ellos en IPFS, blockchain y Hyphanet/Freenet, así como también se documenta el proceso del mismo.

2. Palabras Clave

Distribuido. Sistema. Comunitario. Descentralizado. Aplicación.

3. Abstract

The following work analyzes different ecosystems and technologies that can be used to deploy community web applications in a distributed and decentralized manner.

Using three use cases that illustrate different features: an informational website, a knowledge repository and a real-time messenger, the advantages and disadvantages of deploying each of them on IPFS, blockchain and Hyphanet/Freenet are compared, as well as the process to do so is documented.

4. Keywords

Distributed. System. Community. Decentralized. Application.

5. Introducción

Hoy en día, al querer desplegar una aplicación o sitio web comunitario, lo más común es hacerlo a través de un servicio de alojamiento (AWS, Azure, Google Cloud, entre otros) por la comodidad y facilidad que estas ofrecen, alquilando sus servidores para guardar y procesar datos.

Esto puede llegar a traer problemas para este tipo de aplicaciones. Uno de estos problemas puede ser monetario, ya que muchas veces estas aplicaciones dependen de donaciones o voluntarios para sustentarse, como es el caso de Wikipedia. Como también puede suceder que se encuentre en una zona de censura, lo cual facilita su bloqueo al ser servicios centralizados; entre otros problemas más.

Sin embargo, existen otros ecosistemas alternativos que se asemejan mucho más a la filosofía de estas aplicaciones, y que ayudan a combatir estos problemas. En donde las aplicaciones pueden estar alojadas por sus propios usuarios, donando su computo o espacio, y así logrando una descentralización.

En el siguiente documento presentamos un análisis sobre la infraestructura existente, donde es posible la implementación de plataformas para el despliegue de este tipo de aplicaciones, recabando las bondades y desventajas que cada una tiene.

Para esto se crearon diferentes casos de uso que representan posibles aplicaciones sobre esta metodología alternativa analizando su viabilidad. Entre ellos, se encuentran un sitio web estático, una enciclopedia colaborativa y una aplicación de comunicación en tiempo real.

6. 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.

6.1. Introducción a arquitecturas de red

Comunicar distintas computadoras es un trabajo que requiere coordinación por parte de todas las partes, protocolos para estandarizar la información que se transmite, e infraestructura para poder envíar cada bit de origen a destino. Entonces, se debe diseñar una red coordinada para poder ofrecer los distintos servicios a través de Internet. Para ello, existen dos arquitecturas principales.

Cliente-Servidor

Presente en la gran mayoría de las aplicaciones de Internet, el modelo *Cliente-Servidor* consiste en mantener un nodo central (servidor), quién se encarga de manejar la interacción entre los demás nodos (clientes), y entre clientes y el mismo servidor. Este modelo se clasifica como **centralizado**, debido a que la sub-red de sistemas depende del nodo servidor, y los clientes no tienen manera de comunicarse sin él ante una eventual caída del servidor.

Entre los servicios de Internet más utilizados que utilizan esta arquitectura se encuentra la World Wide Web, el servicio de e-mail (SMTP), el servicio de DNS, entre otros.

Peer-to-Peer

El modelo **peer-to-peer(P2P)** consiste en una red **descentralizada** que tiene distintos nodos (pares) 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 a la vez.

BitTorrent El servicio más utilizado que implementa este modelo es la red de BitTorrent, 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.

6.2. 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 mas utilizada,

Resiliencia Cuando un servidor se encuentra fuera de servicio, toda la red que depende de él no funcionará en tanto no se restaure el servidor. Para contrarrestar esta vulnerabilidad del modelo, se desarrollaron métodos a lo largo de los años. Una manera de evitar que la red se vuelva inoperativa es la de alojar diferentes instancias del servidor en diferentes zonas geográficas. Además, se pueden implementar medidas para evitar la caída del servidor, como los técnicas de balanceo de cargas y fuentes de energía alternativas para evitar eventuales cortes de electricidad.

No obstante, una red peer-to-peer puede ser incluso más robusta. Si hay suficientes pares en la misma y están lo suficientemente dispersos geográficamente, la desconexión de uno de ellos no desactiva toda la red. Esto permite que el modelo peer-to-peer pueda ser resistente a cortes de energía masivos y desastres naturales, lo que lo hace un modelo ideal para servicios prioritarios.

Cabe destacar que en una red de una cantidad limitada de pares, en donde no hay redundancia del contenido que se distribuye, es posible que al desconectar uno de los pares parte del contenido

se vuelva no disponible. Por lo tanto, si bien la red seguirá activa, no tendrá toda la funcionalidad que si puede ofrecer un servidor mientras siga en línea.

Escalabilidad La escalabilidad de una red peer-to-peer aumenta con la cantidad de nodos disponibles, dado que hay más recursos y, en una red bien diseñada, la carga se distribuye equitativamente. En un modelo cliente-servidor, garantizar escalabilidad se puede tornar costoso. Un servidor con mayor capacidad para comunicaciones entrantes y volumen de información tiene hardware de un costo mayor. En casos de aplicaciones de uso masivo puede ser necesario multiplicar la cantidad de nodos servidores para satisfacer la demanda de clientes.

Control del contenido Un servidor, al ser la pieza central de la red a la cuál pertenece, debe soportar múltiples conexiones en simultáneo. Para aplicaciones de alto tráfico, esto requiere una infraestructura que los usuarios suelen no poseer. Una solución es tercerizar el alojamiento de la aplicación servidor en plataformas de Cloud Hosting como pueden ser AWS, Azure, Google Cloud, entre otras. Estos servicios mantienen los servidores de numerosas aplicaciones de Internet, y por lo tanto, tienen la capacidad de modificar, censurar, o remover cualquiera de ellas si así lo desean.

Una red P2P no sufre de estos problemas, ya que por naturaleza los usuarios son quienes la alojan. Por lo tanto, remover contenido de ella resulta mucho mas complejo. Esto evita la censura en zonas donde el acceso a Internet es controlado y/o limitado, pero también puede incentivar a la distribución de contenido ilegal.

Seguridad La seguridad en las aplicaciones cliente-servidor se ha investigado por mucho más tiempo debido a la popularidad de este modelo. Además, al ser centralizado, el propietario del servidor puede bloquear conexiones y eliminar contenido malicioso de su plataforma de forma transparente para los usuarios.

El modelo descentralizado, en cambio, no cuenta con el desarrollo en términos de seguridad. En este caso, el cliente es el responsable de conectarse a redes de confianza, o de hacerlo mediante VPNs (Virtual Private Networks) para mantener el anonimato mientras se integre una red P2P. Sin embargo, cualquiera sea el modelo utilizado por una aplicación, la mayor parte de la seguridad dependerá de que tan segura sea la aplicación.

Persistencia Como varias otras propiedades del modelo cliente-servidor, depende de la integridad del servidor. Si el almacenamiento físico de este se ve afectado, los datos pueden perderse definitivamente. Por esta razón, es común tener un respaldo de los datos de la aplicación en otro disco u otro nodo para evitar la pérdida total de datos.

En una red descentralizada, la persistencia depende de la aplicación utilizada. En el caso de BitTorrent, cada nodo que se conecte y descargue un archivo, podrá compartir ese archivo con otros nodos, y por lo tanto ese archivo contará con una redundancia adicional, la cuál crece a medida que más personas descargan ese archivo. A pesar de esto, en casos donde el archivo es poco compartido, puede volverse inaccesible si los nodos que lo contienen se desconectan de la red.

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, juegos, 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 (Content Delivery Network). 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 Los costos de alojar una aplicación peer-to-peer suele ser nulo, ya que los mismos usuarios de ella son los encargados de proporcionar la infraestructura de la red.

Al contrario, alojar la aplicación en un servidor conlleva tener un servidor disponible, o bien contratar un servicio de web hosting, cuya tarifa suele aumentar a medida que la aplicación escala. En la mayoría de los casos, el costo termina siendo significativo, por lo que una aplicación descentralizada es una opción viable en escenarios en los que no se desee invertir mucho dinero.

6.3. 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.

6.3.1. IPFS

Un suite modular de protocolos para organizar y transferir datos, diseñado con los principios de *content addressing* (recuperación de archivos en base a contenido y no en base al nombre o id) y una red peer-to-peer. Su principal caso de uso es para publicar datos como archivos, directorios y páginas web descentralizadas.[28]

libp2p Colección de protocolos y utilidades para facilitar la implementación de una red peer-to-peer [38]. 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, se necesita saber la dirección del nodo que tiene dicho contenido. El principal protocolo para lograr esto se denomina Distributed Hash Table (DHT) [11]. Es un registro clave-valor distribuido en todos los nodos que soporten este protocolo, que contiene la información necesaria para encontrar el contenido deseado. Cada nodo tiene una parte de esta tabla, y deberá preguntar a otros nodos hasta conseguir la dirección del nodo asociada a la dirección del contenido buscado.

Protocolos de mensaje

Kubo La implementación principal de IPFS es Kubo [37], una solución hecha en Go. Tiene su propio comando en la terminal, llamado `ipfs`, y también es utilizado en los demás frontends 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 Herramienta para orquestar distintos nodos de IPFS, con el objetivo de mantener disponible un contenido en la red de IPFS, aumentando su disponibilidad. Por defecto, cualquier nodo puede modificar la lista de archivos que mantiene el *cluster*.

Clusters colaborativos Son clusters que permiten que usuarios puedan colaborar con su nodo y aumentar la disponibilidad del contenido, sin tener permiso para modificar los archivos que se están manteniendo [29]. Esto es ideal para una aplicación comunitaria debido a esta medida de seguridad que permite al cluster ser apoyado por cualquier usuario de una comunidad.

OrbitDB Base de datos peer-to-peer descentralizada [45]. Utiliza por debajo IPFS para el almacenamiento de los datos, y *libp2p* como soporte para sincronizar cada base de datos entre todos los peers. Es *evetualmente consistente*, lo que significa que modificar una base de datos no asegura que este cambio esté disponible en el resto de los nodos bajo un tiempo específico. Esta herramienta tiene una implementación en Javascript, y otra mayormente abandonada en Go. Sin embargo, un nodo de OrbitDB puede ser utilizado en el entorno de un navegador con los parámetros correctos, lo cuál es útil para crear una aplicación web.

Helia Para crear una instancia de un nodo de OrbitDB se debe pasar como parámetro una instancia de Helia. Helia [19] es la implementación de IPFS para Javascript, y, como OrbitDB, también puede ejecutarse en un navegador. A diferencia de Kubo, Helia tiene un enfoque mucho más abierto, permitiendo la configuración del nodo libp2p utilizado internamente de manera abierta. Esto implica un trade-off claro: más configuración permite un nodo de IPFS que se puede adaptar a diferentes escenarios, pero también implica una mayor complejidad a la hora de lograr un funcionamiento correcto del nodo.

6.3.2. Blockchain

Tecnología basada en una cadena de bloques de operaciones descentralizada y pública. Esta tecnología genera una base de datos compartida a la que tienen acceso sus participantes, los cuáles pueden rastrear cada transacción que hayan realizado.[24]

6.3.3. Alternativas

Freenet/Hyphanet Programa de código abierto para compartir datos peer-to-peer con enfoque en la protección de la privacidad. Opera en una red descentralizada, promocionando la libertad de expresión facilitando la anonimidad de los datos compartidos y eludiendo la censura.[17][25]

7. Problema detectado y/o faltante

Los servicios actuales que proveen de infraestructura tienden a ser muy costosos para las pequeñas comunidades que necesitan tener un servicio con alta disponibilidad, accesible para todos y barato de escalar. Actualmente tampoco existe un estándar para aplicaciones cuyo stack sea completamente distribuido usando peer-to-peer.

7.1. Costos

La inversión para el mantenimiento de servidores puede ser un obstáculo a la hora de proveer una red a sus usuarios cuando se trata de una aplicación pequeña o startup. En estos casos, es común sacrificar la escalabilidad en pos de mantener los costos del alojamiento de la aplicación bajos.

En el otro extremo del espectro, se encuentran las aplicaciones en declive. Debido a la falta de incentivos financieros para justificar el mantenimiento, muchas veces la solución es desconectar

los servidores definitivamente. Esto es especialmente frecuente en videojuegos multijugador que no cuentan con la capacidad de crear servidores dedicados. En tales situaciones, la empresa propietaria puede desactivar los servidores, lo que resulta en que el videojuego se vuelva parcialmente o completamente inutilizable. [4]

7.2. Interrupciones del servicio

Dada la naturaleza del modelo cliente-servidor, no es infrecuente que estas redes se vuelvan inaccesibles. Esto puede ocurrir deliberadamente por temas de mantenimiento, o accidentalmente debido a modificaciones en la aplicación del servidor durante el mantenimiento o actualización de la misma.

Uno de los episodios recientes de mayor revuelo ocurrió el 4 de Octubre de 2021, día en el que la familia de aplicaciones de Meta -Whatsapp, Facebook e Instagram -, estuvo fuera de servicio por seis horas. Esto ocasionó que mas de 3500 millones de usuarios se vieran afectados. En un articulo posterior, Meta reveló que la causa se debió a una falla en la herramienta de auditoría de los comandos que se envían a la red global de datacenters de Meta, la cual evitó que se pudiera detener el comando que desconectaba los servidores de la red. [36]

7.3. Zonas de censura

Es común que aplicaciones o sitios web comunitarios sean sometidos a su censura. La centralización de los servicios ayuda a que sea mucho más fácil bloquear su acceso, dado que es más fácil identificar y bloquear un único punto de acceso. En contraste, los sistemas descentralizados, suelen ser más resistentes porque no dependen de un servidor o servicio central que pueda ser intervenido fácilmente.

Uno de los casos más conocidos y vigentes es la censura de Wikipedia. Donde algunos gobiernos bloquean su acceso en un determinado lenguaje y otros en su totalidad. [5]

8. Solución implementada

Se implementaron tres casos de uso sobre los distintos ecosistemas a analizar. Se detalla el camino recorrido para llegar a cada implementación, haciendo uso de las distintas herramientas presentes en cada ecosistema para lograr cumplir con los requisitos funcionales presentados a continuación. Por último, presentamos un análisis cualitativo y cuantitativo de las ventajas y desventajas de cada caso dependiendo del ecosistema.

8.1. Casos de uso

8.1.1. Sitio web informativo

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

Requisitos funcionales

- **Landing page del proyecto**: sitio web informativo donde se presente información sobre este proyecto, los casos de uso y sus ecosistemas de despliegue.

8.1.2. Repositorio de conocimiento

Esta 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.

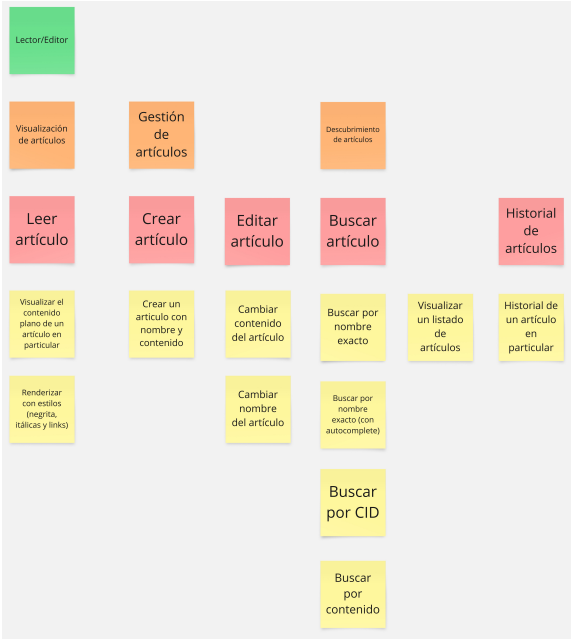

Figura 1: *User Story Map* del repositorio de conocimiento

Arquitectura 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* [39] 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.

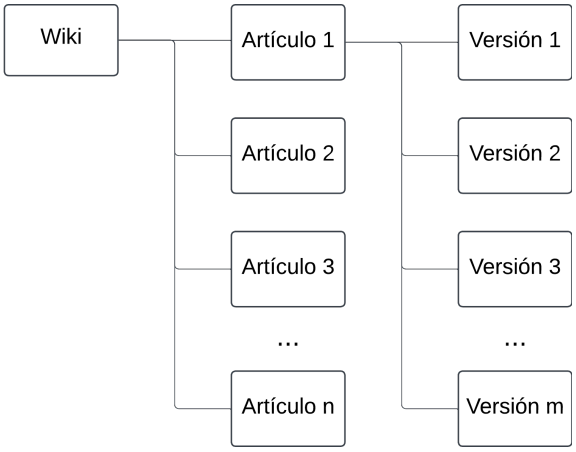

Figura 2: Arquitectura general del repositorio de conocimiento.

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 [**wiki-edits-stats**]. 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* [10], 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 [44], 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.

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.

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 [55], 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.

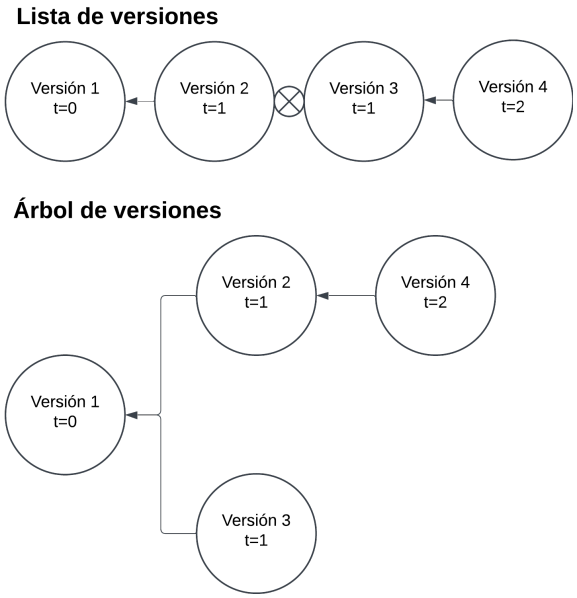

Figura 3: Ambas maneras de almacenar versiones. En este caso, `t` es un momento en el tiempo en el que todas las bases de datos están sincronizadas. En el primer caso, cuando `t=1` en ambas versiones, significa que hay un conflicto ya que ambas se basan en la misma versión padre. En el árbol, ese problema 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 [41].

```
type VersionID = string;

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

Figura 4: Propiedades de una `Version`

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.

8.1.3. 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.

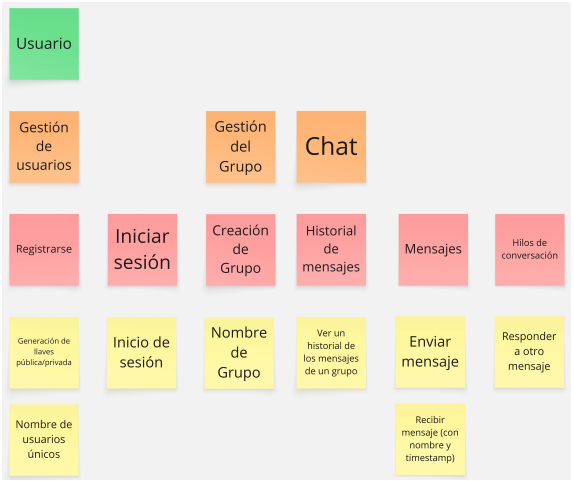

Figura 5: *User Story Map* del mensajero en tiempo real

Arquitectura 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 6: Propiedades de un `ChatMessage`

8.2. Proceso de descubrimiento

Durante la implementación de los casos de uso para cada ecosistema, se fue desarrollando un mejor entendimiento de lo que se estaba creando. Logrando así generar distintas abstracciones que representan la infraestructura general para la implementación de los casos de uso.

Para cada ecosistema hay 2 principales infraestructuras en acción. La infraestructura de despliegue y la infraestructura de aplicación.

La **infraestructura de despliegue** es la encargada del *hosting* de una aplicación web. Con esta se distribuye y permite el acceso a lo que es el *front-end* de las aplicaciones, como también el código para el funcionamiento de la aplicación, si se trata de una aplicación no estática. Cualquier aplicación que desee ser accedida por la web va a hacer uso de esta infraestructura, como es el caso del sitio web informativo.

La **infraestructura de aplicación** es la encargada de la lógica de la aplicación, como es el almacenamiento de datos, como la conexión entre pares. Aplicaciones que requieran de mantener un estado y permitir la modificación de parte de los usuarios van a necesitar hacer uso de esta infraestructura, como es el caso del repositorio de conocimiento y mensajero en tiempo real.

Al lograr encontrar estas abstracciones, se permite que generar nuevos casos de uso sea mucho más fácil, ya que la mayoría de la lógica sobre el el ecosistema se encuentra encapsulada dentro de ellas, logrando que el desarrollo de un nuevo caso de uso se concentre únicamente en sus requisitos y no en el ecosistema en el que se encuentra.

A continuación se va a explicar como se componen y funcionan ambas infraestructuras en cada ecosistema.

8.3. IPFS

8.3.1. Infraestructura de despliegue

En IPFS, es posible desplegar una aplicación web subiendo un directorio con todos los archivos estáticos necesarios para el funcionamiento en un navegador, incluyendo el código necesario a nivel aplicación. Esto se puede realizar manualmente mediante cualquier cliente de IPFS, como Kubo[37] o Helia[19], y devuelve un *content identifier* (CID) [30] que representa esa versión de la aplicación.

Cualquier usuario puede publicar su sitio web en la red de IPFS a través de un nodo local de manera gratuita y poco tiempo. IPFS provee un tutorial en su página de cómo realizarlo [20]. A continuación se detallará las implicaciones que tiene desplegar una aplicación web de esta manera, y que alternativas existen para publicar en IPFS.

Al subir un archivo —por ejemplo, código HTML— su contenido se inserta en una función de hash, y así se obtiene su CID. 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. Esto se debe al *garbage collector* [31] implementado por IPFS, que desecha datos para liberar almacenamiento de forma arbitraria. Por esta razón existe el concepto de [48]: *"pinear"* un archivo o directorio significa instruir al nodo IPFS para que trate dicha información como esencial y, por lo tanto, no lo descarte.

No obstante, pinear un archivo o directorio no asegura su disponibilidad indefinida en el tiempo, ya a que esta depende de que el nodo que lo tiene pineado esté activo, o de que otros nodos que hayan accedido al archivo y aún lo tengan en su caché. Para mejorar la disponibilidad de un archivo, lo ideal es que varios nodos pineen el contenido, de modo que otro nodo que desee obtener el contenido pueda hacerlo desde cualquiera de ellos.

Para lograr que el contenido persista en la red sin necesidad de que el nodo local esté activo, existen opciones para delegar el pineo del archivo o directorio. Existen servicios de pinning y clusters colaborativos, que actúan pineando los archivos en múltiples nodos, aumentando no solo su disponibilidad sino también su distribución, y por ende logrando un acceso más rápido al contenido.

Servicios de *pinning* La manera más fácil de asegurarse que los datos estén disponibles y se persistan es usar un servicio de *pinning* [33]. Estos servicios cuentan con varios nodos que pinnean archivos. De esta manera, ya no es necesario contar con un nodo local que los aloje. Algunos ejemplos de servicios de pinning incluyen Fleek [16] y Pinata [47].

Desde el punto de vista de las aplicaciones estrictamente comunitarias, estos servicios no van de la mano con su filosofía. Por un lado, los servicios de *pinning* tienen un modelo gratis con funcionalidad limitada o capacidad de almacenamiento limitado. Por otro lado, se depende de estos servicios, lo que en esencia centraliza el proceso de despliegue de la aplicación o sitio web. Si por algún motivo el servicio dejara de pinear los archivos, estos pueden dejar de estar disponibles en la red IPFS, e incluso pueden perderse por completo. Esto rompe completamente con la naturaleza de aplicaciones descentralizadas y pasa a tener una centralización tercerizada similar a utilizar un *cloud hosting*.

Clusters colaborativos Un *cluster* es un grupo de nodos de IPFS que actúan en conjunto para pinear un contenido. Funcionan sincronizando su *pin set*, o sea, su lista de archivos y directorios pineados en un momento dado. Un cluster *colaborativo* sigue esta premisa, pero permite que los usuarios puedan colaborar con su nodo local para el pineo de la aplicación sin tener la posibilidad de modificar los archivos, la cuál es delegada a nodos especiales que tienen la capacidad de orquestar el cluster en conjunto. Así, se logra que la misma comunidad mantenga en servicio el mecanismo de despliegue de la aplicación, lo cuál es acorde a la filosofía de aplicaciones comunitarias.

Actualmente, esta alternativa es poco explorada, por lo tanto no existe una forma fácil de creación, seguimiento y descubrimiento de estos clusters. IPFS cuenta con una página con clusters conocidos con los cuales se puede colaborar [29], pero la cantidad es limitada.

Por otro lado, el principal problema es que los clusters obligan a los nodos a "pinear"la totalidad de sus archivos, lo cuál puede significar un uso excesivo de almacenamiento necesario para colaborar. Hacer sharding sobre el pin set, o sea, pinear parte del contenido de un cluster, es posible utilizando los parámetros de replicator_min_max al agregar un pin, que fijan un límite mínimo y máximo sobre la cantidad de nodos que tienen ese pin. Sin embargo, no es recomendado para clusters colaborativos debido a la falta de *proof of storage* [21] [7]. Esto se debe a que, debido a la manera en la que fue diseñada la arquitectura de IPFS, un nodo no confiable puede falsificar la lista de archivos que está pineando, por lo que hay una posibilidad de que una parte del contenido no esté en ninguno de los nodos, y por ende el contenido esté incompleto.

Acceso y mutabilidad Para buscar un contenido, un nodo de IPFS realiza una búsqueda a través de su CID, el cual es único. Debido a que es único, el CID cambiará si el contenido del sitio web o aplicación web cambia, ya que el contenido será distinto. Esto vuelve el proceso de despliegue altamente impráctico, ya que se necesitaría compartir un nuevo CID cada vez que se actualice una página.

Este problema puede ser resuelto con la ayuda de *punteros mutables*. Estos punteros son un objeto de IPFS que apunta a un CID determinado, previamente elegido por el usuario. El CID al que apunta el puntero puede ser cambiado, por lo tanto permiten compartir la dirección del puntero una única vez y actualizar el CID al cuál apunta cada vez que se haga un cambio.

IPNS InterPlanetary Name System (IPNS) [27] es un sistema que permite crear punteros mutables y obtener su dirección en forma de CIDs conocidos como *names* o *nombres de IPNS*. Estos nombres de IPNS pueden considerarse como enlaces que pueden actualizarse, conservando al mismo tiempo la verificabilidad del content addressing.

Un nombre de IPNS es un hash de una [1] clave pública. Está asociado a un *IPNS record* [35] que contiene la ruta a la que se vincula, entre otra información. El titular de la clave privada puede firmar y publicar nuevos registros en cualquier momento.

Es posible utilizar IPNS con uno de estos posibles enfoques:

- **Consistencia:** garantizar que los usuarios siempre resuelvan el último registro de IPNS publicado, a riesgo de no poder resolverlo.
- **Disponibilidad:** resolver un registro de IPNS válido, a costa de potencialmente resolver un registro desactualizado -o sea, con un CID previo.

El registro IPNS se encuentra a través de la **Distributed Hash Table** (DHT) [11]. Todos los nodos de IPFS participan alojando colaborativamente el contenido de la DHT. Por lo tanto, el DHT actúa como un "directorio"descentralizado, donde la clave pública es un identificador. Esta tabla ayuda a localizar el registro IPNS que apunta al contenido deseado, entre otras funciones. Para entender mejor cómo IPNS funciona se puede consultar la documentación de IPFS.

IPNS es una buena forma de obtener mutabilidad dentro de IPFS. 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 names sean hashes alfanuméricos, y no nombres legibles o memorables para humanos, representa una dificultad adicional a la hora de alojar un sitio web al cuál los usuarios puedan acceder fácilmente. A continuación se analizará dos alternativas para solucionar este problema.

```
/ipns/k51qzi5uqu5dhkdbjdsauuyk5iyq82uzpjb0is3x6oy9dcmmr8dbcezv7v9fya
```

Figura 7: Ejemplo de la dirección de un nombre de IPNS

DNSLink IPNS no es la única forma de crear mutable pointers en IPFS. DNSLink [13] utiliza registros *DNS TXT* para asignar un nombre DNS (por ejemplo, un dominio) a una dirección IPFS o a un *IPNS name*. Como uno puede editar sus registros DNS, puede usarlos para que siempre apunten a la última versión de un objeto en IPFS.

DNSLink actualmente es mucho más rápido que IPNS, utiliza nombres legibles por humanos y también puede apuntar a nombres de IPNS. A pesar de ello, tiene un problema muy fundamental y es que se utiliza el protocolo **DNS**, el cual tiene claras deficiencias con la filosofía de aplicaciones comunitarias.

La más importante es que, aunque DNS tenga claras ventajas, como ser un sistema distribuido y escalable, es también un sistema algo centralizado. Las autoridades centrales como *ICANN* gestionan las raíces del DNS. Esto hace que un registro DNS sea fácil de censurar, a nivel de registrador como también a nivel *ISPs*.

ENS **Ethereum Name Service (ENS)** [14], es el protocolo de nombres descentralizado que se basa en blockchain *Ethereum*. Funciona de manera similar a DNS, en el sentido de que los nombres ENS resuelven a nombres legibles para humanos. Como esto se computa en la blockchain de Ethereum, es seguro, descentralizado y transparente. Está diseñado específicamente para traducir

identificadores como direcciones de billeteras de criptomonedas, hashes, metadata, entre otros, incluyendo direcciones de IPFS.

Es posible configurar un registro ENS para que se resuelva automáticamente la dirección IPNS, proporcionando nombres legibles para humanos que son más fáciles de compartir y acceder, y solucionando 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 siempre se va a apuntar al mismo nombre de IPNS.

Cabe aclarar que adquirir un dominio ENS tiene un costo, que depende de varios factores [15]:

- El largo del nombre. Un nombre con menos caracteres tiende a tener un valor mayor.
- Cuán reciente expiró la licencia del dominio. Si un dominio expiró recientemente, se le aplica un precio *premium* que decrece con el tiempo. Un dominio con mayor uso aumenta su precio.
- El valor del gas actual, que depende de la congestión de la blockchain.

Acceso desde un navegador Por último, se necesita una manera de acceder a los archivos alojados en IPFS. En navegadores que soportan IPFS y ENS —como Opera [2] y previamente Brave [3]— se puede acceder directamente. En la mayoría de los navegadores, sin embargo, esta no es una opción. Para lograr un mayor alcance que incluya estos navegadores, se requiere el uso de una *IPFS gateway* [32].

Una IPFS gateway es un nodo que recibe requests HTTP que contienen una dirección de IPFS, busca el contenido en la red de IPFS, y lo devuelve en una HTTP response. Esto es útil tanto para archivos como para directorios. Algunas gateways tienen la funcionalidad de mostrar una página web de manera correcta cuando un directorio tiene la estructura indicada. Esto nos es particularmente útil para poder mostrar una página moderna de la misma manera que se haría utilizando un servidor HTTP.

Una lista de gateways disponibles puede obtenerse utilizando el Public Gateway Checker [49] proporcionado por IPFS.

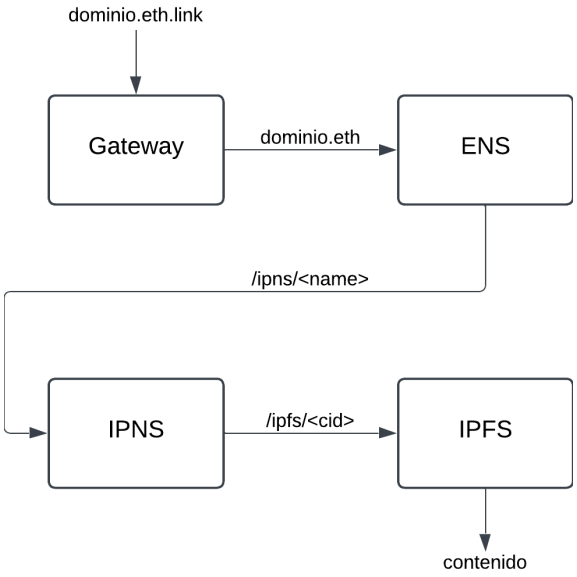

Figura 8: Mapa de la traducción de un dominio al contenido de IPFS

Despliegue continuo En un proyecto de aplicación web centralizada, es común automatizar el proceso de despliegue con cada cambio que se realiza. Normalmente este proceso se activa con cada nuevo commit en una rama de Git especifica, e incluye todas las etapas necesarias para convertir el

contenido de un repositorio Git en código estático listo para ser desplegado. También puede incluir más pasos que incluyan actualizaciones en el backend.

Yendo al caso específico de aplicaciones web comunitarias, el script debe ser ejecutado en los nodos confiables, ya que una *Github action* no puede utilizar un nodo IPFS que requiera puertos abiertos. En este tipo de aplicaciones, al tener una jerarquía mayormente horizontal, no hay un servidor central que orqueste esta actualización, sino que se necesita que cualquier nodo confiable pueda actualizar su contenido e instruir a los nodos colaborativos para actualizar su contenido de igual forma. Todo esto debe ser posible incluso cuando los nodos no reciben la actualización al mismo tiempo, es decir, no debe haber *race conditions*.

Una forma de lograr esto es, por ejemplo, utilizar un algoritmo de elección de líder u otro algoritmo distribuido para elegir el nodo responsable de indicar el nuevo contenido a pinear al resto de nodos en el cluster. Sin embargo, esta manera de realizar la actualización implica una capa adicional de complejidad que no es necesaria debido a la naturaleza de IPFS.

Como ya se ha mencionado, si dos nodos suben el mismo contenido, obtendrán el mismo CID. Esto puede ser utilizado para que cualquier nodo confiable pueda actualizar el contenido y el nombre de IPNS independientemente del resto de los nodos confiables. Cuando se detecte un cambio nuevo, el nodo puede obtener el código estático, y acto posterior, indicar al resto de los nodos del cluster que pineen el CID especifico. En el caso de que sea el primer nodo en detectar el cambio, deberá instruir al resto del cluster para que dejen de pinear el CID antiguo. En el caso en que otro nodo haya detectado la actualización antes, no deberá actualizar ningún pin del cluster debido a que el mismo CID ya va a estar presente en la lista de pins.

Compilación Las herramientas de compilado no siempre son deterministas en los archivos compilados que genera. Next.js, por ejemplo, genera diferentes archivos estáticos en dos compilaciones basadas en el mismo código fuente. Esto es un problema para el enfoque propuesto, debido a que si dos nodos compilan el mismo código, el CID puede ser diferente. Para mitigar esto, se decidió hacer uso de un *hook* que compile el código con cada *commit* en la rama principal una única vez por cambio realizado. De esta manera, los nodos confiables pueden detectar el cambio en la rama utilizada para alojar los archivos estáticos, y hacer *pull* sobre esos archivos y, por lo tanto, obtener un mismo CID.

Jerarquía 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 cluster colaborativo compuesto por nodos 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.

Si bien el objetivo es lograr una aplicación comunitaria, se debe establecer de todas formas distintos rangos para proteger el proyecto de ataques. Como el nombre de IPNS cambiará a lo largo del tiempo en tanto se realicen cambio en el proyecto, se vuelve necesario seleccionar un grupo de nodos que se les confíe con tal fin. Esto se debe a que, de lo contrario, un posible atacante podría modificar el registro para invalidarlo o cambiar el contenido al que apunta. Por la misma razón, no cualquier nodo dentro del cluster debe ser capaz de cambiar el *pin set*, o lista de CIDs a los cuáles cada nodo del cluster pinea.

IPFS Cluster tiene en cuenta esto, y hace la distinción entre un nodo *trusted* y un nodo *follower* para su implementación de clusters *colaborativos*[6]. 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 cluster, ya que sin estos nodos no se podrá modificar el contenido. Esto no supone una desventaja ni tampoco hace que la solución se vuelva centralizada en el grupo de nodos confiables actual, debido a que los usuarios de la comunidad pueden crear su propio grupo 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 pinear los archivos establecidos por los nodos confiables, y actualizar su *pin set* cuando se lo indique. Al igual que los nodos confiables, debe pinear 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 en simultáneo. Esto previene un posible *single point of failure* y asegura que el cluster siempre se encuentre en un estado válido.

`service.json` Para que un usuario pueda conectarse y contribuir como colaborador a un cluster, la herramienta de terminal `ipfs-cluster-follow` [34] requiere una dirección de IPFS de la cuál obtener el archivo `service.json` [8]. Este archivo de configuración contiene todos los datos necesarios para que un colaborador pueda unirse. Además, está sujeto a modificaciones, debido a que el archivo contiene las *multiaddresses* [42] de cada nodo confiable en forma de lista, por lo que agregar o remover un nodo confiable implica modificar el archivo. Es por esto que el proceso de despliegue también debe incluir este archivo. Desde la detección de una actualización en un repositorio de Git que lo contenga, el pineo del nuevo `service.json` al cluster, hasta la actualización de un nombre de IPNS que pueda distribuirse a los usuarios que quieran colaborar.

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

Figura 9: Ejemplo de una *multiadress* posible que utiliza el protocolo QUIC.

Limitaciones Este enfoque, a cambio de ofrecer una solución comunitaria y descentralizada, tiene desventajas o aspectos a mejorar:

Necesidad de tener nodos confiables Estos nodos van a ser los encargados de administrar el cluster, y actualizar el IPNS. La distinción entre nodos confiables y nodos colaborativos es necesaria para evitar que un potencial atacante pueda modificar el CID al que apunta el nombre de IPNS, o modificar el contenido que pinea el cluster colaborativo.

Actualización del contenido Por cada cambio que se realice en el directorio de la página, se deberá pinear el nuevo contenido al cluster, y por lo tanto todos los colaboradores tendrán que obtener todo el directorio nuevamente. Esto puede claramente volverse costoso con contenido de tamaño considerable.

Cache de IPNS El parámetro TTL de IPNS indica cuanto 'vive' un valor asociado a un nombre de IPNS en la cache de un nodo antes de forzar a este a volver a buscar el valor en la DHT. El problema que tiene esto es que, si se pone un valor muy elevado, un nodo gateway no buscará la actualización hasta que se cumpla el periodo y por lo tanto el registro de IPNS no se actualizará. Por otro lado, si se elige un valor muy corto, siempre se buscará el valor en la DHT, generando latencia al no utilizar el cache disponible. Pero a su vez, el nombre de IPNS en un nodo siempre tendrá la última versión que encuentre.

Claves privadas compartidas Cómo la actualización de un nombre de IPNS está firmada con una clave privada, todos los nodos confiables deberán tener la misma clave para poder potencialmente actualizar el registro IPNS y así evitar tener un único nodo con esa responsabilidad. Esto elimina un punto de falla único, pero aumenta las chances de que esa clave privada llegue a manos de un posible atacante.

Apertura de puertos IPFS Cluster utiliza el puerto 9096 para la comunicación entre nodos, el cual se tiene que abrir para un correcto funcionamiento. Esto puede suponer un esfuerzo adicional para usuarios que deseen colaborar.

Implementación Una vez explicado el análisis inicidal y las decisiones que se tomaron para poder lograr un servicio que automatice y facilite parte del despliegue de una aplicación web, se detallará la solución realizada para el nodo confiable. El resultado es una herramienta que se puede levantar utilizando un comando, y automáticamente publique el contenido ubicado en el repositorio de Git dado, encargándose de mantenerlo disponible, de orquestar el *pin set* del cluster, y detectar cambios. El repositorio se puede encontrar en el repositorio de Github [52].

Arquitectura general La herramienta está compuesta por tres contenedores:

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

Todos los contenedores están orquestados mediante Docker Compose. El contenedor watcher está basado en Alpine Linux y utiliza scripts de shell portables. La comunicación entre contenedores se realiza mediante sus respectivas APIs HTTP [23] [22].

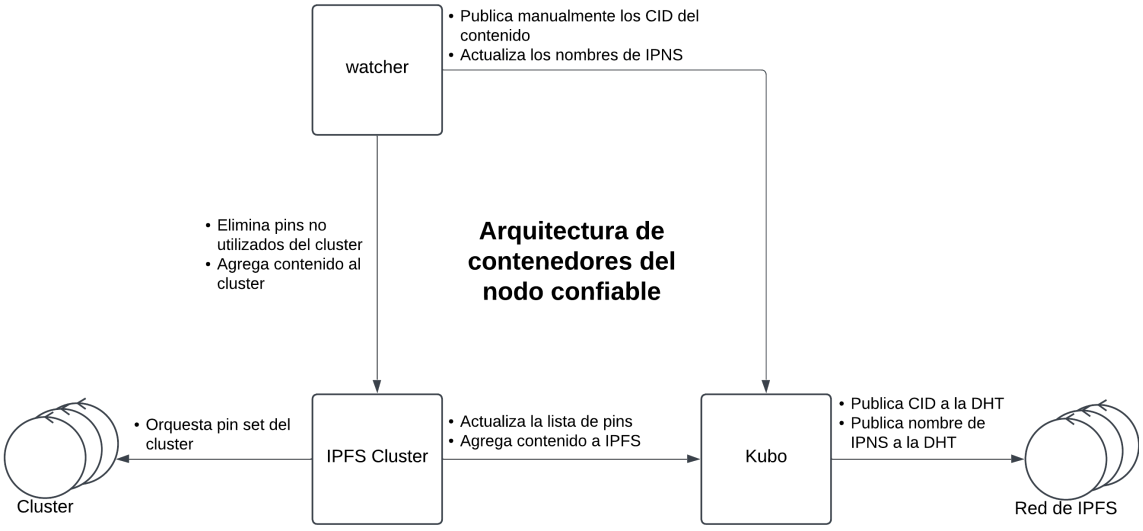

Figura 10: Mapa de interacciones entre los contenedores del nodo confiable

Funcionamiento del Watcher Este módulo del nodo confiable utiliza Git para comparar el último commit de la rama remota contra una copia local que se clona cada vez que se inicia. De esta manera, puede detectar cuando un nuevo cambio ocurre (tanto en el contenido como en `service.json`), e iniciar el proceso para obtener el nuevo cambio y desplegarlo. Dicho proceso se compone de los siguientes pasos:

1. Subir el contenido y el `service.json` al Cluster, y obtener ambos CIDs.
2. En base a los CIDs obtenidos, publicar ambos manualmente utilizando Kubo.
3. Esperar a que todos los nodos dentro del cluster hayan pinneado los nuevos CIDs.
4. Actualizar los dos nombres de IPNS para que apunten a los nuevos CIDs.
5. Eliminar los pins antiguos del cluster.

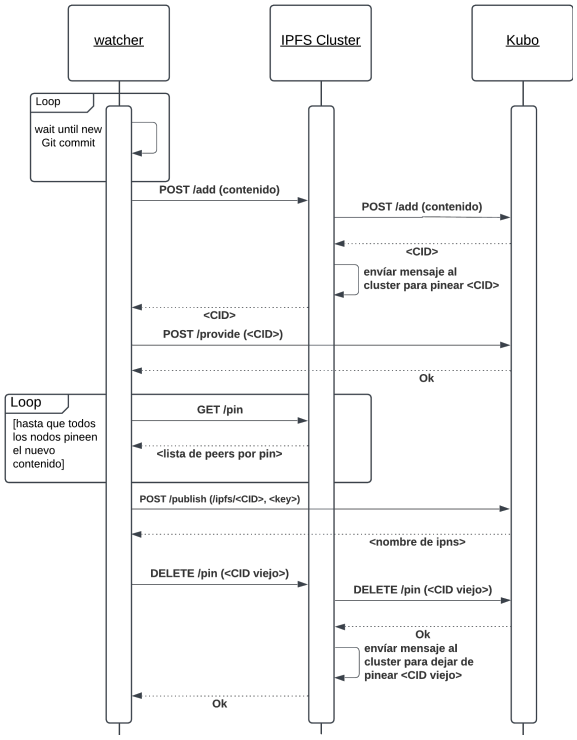

Figura 11: Diagrama de secuencia para el caso en que watcher detecta un cambio. Notar que para mayor claridad se omite los pasos para desplegar el nombre de IPNS del `service.json`, al ser exactamente los mismos que en el caso de un contenido.

Disponibilidad En el proceso mencionado para desplegar los cambios, existen dos factores que pueden afectar a la disponibilidad del contenido luego de recibir una actualización.

Por un lado, el nombre de IPNS puede no haberse actualizado en todos los nodos de la DHT, lo que provoca que algunos nodos apunten a la versión anterior del contenido. Esto se soluciona asegurándose de publicar el nuevo valor del nombre de IPNS **antes de instruir al cluster** para que deje de pinear la versión anterior.

Por otro lado, el contenido nuevo puede no estar disponible inmediatamente, ya que la publicación del CID en la DHT por parte del cluster se realiza de manera asíncrona. Para solucionar esto, se optó por publicar manualmente el CID con Kubo, de forma secuencial, antes de actualizar el nombre de IPNS. La desventaja de este enfoque es el tiempo adicional requerido para publicar el contenido, a cambio de garantizar su disponibilidad en todo momento, ya sea en su versión actual o en la nueva.

Persistencia de la identidad del nodo Sabiendo que para ser un nodo confiable se debe tener su multiaddress en el archivo de `service.json`, es conveniente mantener el mismo PeerID a lo largo del tiempo y en distintas ejecuciones de la herramienta. Para ello, se debe indicar una *identidad* que consiste de un PeerID, y una clave privada. Esto asegura que el nodo siempre se inicie con la misma identificación.

Gestión de claves de IPNS Debido a la naturaleza de IPNS, un nombre solo puede ser modificado por un nodo que posea una clave privada determinada. Por ello, todos los nodos confiables deben tener las mismas clave privada de IPNS, una para el contenido y otra para `service.json`.

Para facilitar la inicialización, la herramienta provee un script que ayuda a generar la configuración y obtener los parámetros necesarios paso a paso. Esto incluye una identificación para el nodo, claves para IPNS, las direcciones de los repositorios de Git, y la IP pública necesaria para

conectar los nodos a la red de IPFS.

Integración con Git La manera en la que el contenedor *watcher* puede detectar un cambio en el repositorio es consultando el repositorio remoto de Git cada minuto para identificar un cambio realizado y accionar el script de despliegue. Se requiere que el repositorio del contenido sea público, ya que la identificación por SSH o usuario y clave no están disponibles fácilmente dentro de un contenedor. De todas maneras, el contenido o archivos estáticos en el caso de una aplicación web ya son públicos por naturaleza, y debido al enfoque comunitario dado, que un repositorio necesite ser público no representa una restricción apreciable.

Resultado La solución implementada logra automatizar el despliegue y la publicación de contenido en IPFS de forma confiable, simplificando muchos aspectos de IPFS y los clusters colaborativos. Mediante un comando `make up` se levanta un nodo confiable que automáticamente puede desplegar y mantener actualizado el contenido que se desee. Cabe destacar que, si bien el enfoque está diseñado para aplicaciones web, esta herramienta permite el despliegue de cualquier tipo de contenido, como repositorios, documentación, etcétera.

Combinando esta herramienta junto con un dominio ENS y un gateway con el cuál acceder al contenido, se obtiene una aplicación web cuyo uso es equiparable a la de un servidor HTTP moderno, sin diferencias perceptibles para el usuario, y de manera comunitaria, descentralizada, y económica.

8.3.2. Infraestructura de aplicación

Para aplicaciones que requieran mantener un estado y permitir que usuarios puedan modificarlo, no es suficiente con la infraestructura que explicamos anteriormente, ya que no hay una noción de estado y solo se le permite cambiar su contenido a los dueños de lo que se despliega. Es por eso que es necesaria otra infraestructura, la cual nos provea de esas necesidades.

Esta infraestructura surgió en base a un extenso desarrollo del cual nos ayudó a entender y encontrar la abstracción de lo que se estaba creando. Al principio del desarrollo, esta era gran parte de la arquitectura del primer caso de uso no estático que realizamos, el repositorio de conocimiento, para luego convertirse en una implementación propia, la cual llamamos **AstraDB**.

A continuación pasaremos a explicar cómo es la arquitectura que compone a AstraDB, cómo fue su evolución y que decisiones se tomaron a lo largo de su desarrollo, como también cómo podemos hacer uso de ella para fácilmente crear las aplicación que venimos a analizar, aplicaciones comunitarias, distribuidas y descentralizadas dentro del ecosistema de IPFS, tal como lo son el repositorio de conocimiento y el mensajero en tiempo real, las cuales hacen uso de AstraDB para su funcionamiento.

Etapa de investigación Al comenzar con el desarrollo del repositorio de conocimiento nos encontramos con un desafío, cómo podemos lograr que una aplicación dentro del ecosistema de IPFS pueda tener, modificar y guardar un estado.

Dada la naturaleza de IPFS, como explicamos anteriormente, no está pensado para alojar cambios en tiempo real. El modelo de direccionamiento por contenido implica que cualquier modificación genera un nuevo identificador (CID), lo que resulta inconveniente para actualizar un recurso directamente sin mecanismos adicionales. Por esta razón, utilizar únicamente el conjunto de protocolos que IPFS ofrece no nos resulta conveniente para aplicaciones dinámicas.

Como sucede con un caso de uso muy similar al repositorio de conocimiento que queremos implementar, el proyecto de **Distributed Wikipedia Mirror**[12], el cual consistió en poner una versión de wikipedia en IPFS, únicamente funciona como versión Read-Only, y con snapshots manuales, lo cual es totalmente posible de hacer con la infraestructura que explicamos anteriormente.

Es por esto que resulta de un verdadero desafío lograr que una versión Read-Write sea posible sin sacrificar los principios de decentralización que IPFS nos provee. Para abordar esta limitación

nos llevó a buscar herramientas complementarias dentro del ecosistema y ahí fue cuando nos encontramos con **OrbitDB**[45].

Representación de los datos OrbitDB es una base de datos peer-to-peer, distribuida y sin un servidor central. Utiliza IPFS para el almacenamiento de datos y **Libp2p**[38] para sincronizar automáticamente las bases de datos con otros peers. Es una base de datos **eventualmente consistente** que utiliza Merkle-CRDTs para escrituras y fusiones de base de datos libres de conflictos, lo que hace que OrbitDB sea una excelente opción para aplicaciones p2p y descentralizadas.

OrbitDB ofrece varios tipos de bases de datos para diferentes modelos de datos y casos de uso. Algunos que se asemejan más a bases de datos convencionales, como puede ser un simple key-value y otros más distintos como puede ser uno de eventos secuenciales.

Uno de los requisitos de la infraestructura desarrollada es lograr una verdadera descentralización, significando que no haya una entidad o persona con mayores permisos sobre el resto, esto se traduce, en parte, a permitir que cualquiera que quiera pueda crear y/o modificar la base de datos sin ninguna restricción. Para lograr esto, OrbitDB nos permite indicar que cualquier nodo tenga permiso de edición sobre una base de datos. El problema es que esto también permitiría que cualquier usuario pueda eliminar información de la base de datos y es algo que no nos podemos permitir y es esta la principal razón por la cual no podemos utilizar un tipo como key-value, o documentos y necesitamos otro que se adecúe más a nuestro caso.

OrbitDB nos provee de otro tipo de base de datos, el tipo de events. Un tipo de base de datos el cual es inmutable (append-only), un log transversal que muestra un historial que se puede recorrer. Sin embargo surgen nuevos desafíos.

Gracias a que es un tipo append-only ya no se tiene el problema de posible perdida de información, ya que un usuario solo puede agregar un nuevo valor a la base de datos y no eliminar valores agregados previamente, sin embargo nos surge otro problema, como representamos la información con una base de datos que solo se puede agregar?

La solución es separarnos de la idea de que se tiene que tener una única base de datos que albergue toda la información. Al pensarlo desde el punto de vista del repositorio de reconocimiento, podemos representar cada articulo como su única base de datos de eventos y cada valor que se le agrega puede ser el cambio que se le hizo al articulo y al leer su historia se puede reconstruir a su última versión.. Esto resulta de una práctica muy común en estas arquitecturas distribuidas.

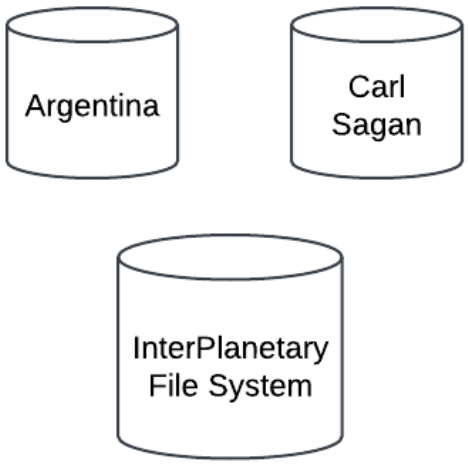

Figura 12: Representación de cada articulo como su propia base de datos.

De esta manera ya tenemos una representación para los artículos, pero faltaría una forma de

saber que artículos existen actualmente. Es por eso que tenemos que agregar una última base de datos, también de eventos, que represente la wiki en si, con los nombres de los artículos existentes. De esta manera un usuario puede saber, al acceder a esta base de datos representando un repositorio de artículos, cuales son los artículos que existen y solo acceder a la base de datos correspondiente a ese articulo, sin necesidad de replicar la totalidad de los artículos existentes, algo que sería inviable.

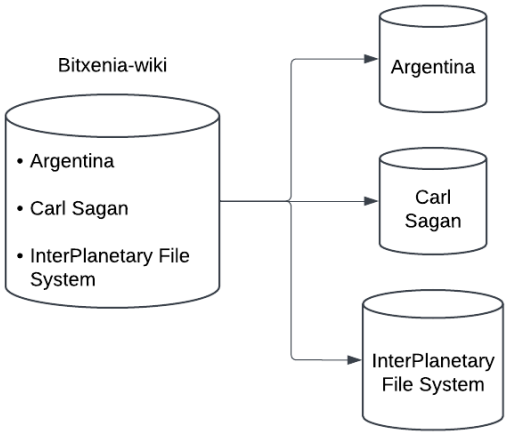

Figura 13: Representación del repositorio de artículos como base de datos.

Ahora bien, como podemos acceder a la base de datos. En OrbitDB las bases de datos tienen una dirección que las identifica, formada en su creación. Esta dirección está conformada por el nombre de la base de datos, su tipo y su *Access Controller*. Este último sirve para indicar quien tiene acceso y permisos sobre la base de datos. Como en nuestro caso estamos permitiendo que todos tengan permiso, el controller va a ser siempre el mismo. Esto nos es importante, ya que significa que al crear la base de datos siempre vamos a tener una misma dirección y es por eso que con solo saber el nombre de un articulo podemos saber la dirección de la base de datos que la compone, no necesitamos guardarnos ninguna dirección especifica. Lo mismo sucede con la base de datos representando la wiki, con saber el nombre de la wiki podemos acceder a ella.

```
/orbitdb/zdpuAmrcSRUhkQcnRQ6p4bphs7DJWGBkqczSGFYynX6moTcDL
```

Figura 14: Ejemplo de un *address* de una base de datos de OrbitDB.

Colaboradores En orbitdb, al ser una base de datos peer-to-peer, distribuida y sin un servidor central, significa que la base de datos existe en cada peer que la componga, por lo tanto conectarse a una base de datos significa replicar la totalidad de su información de otros peers que nos la provean. De no estar esos peers, significaría que la base de datos no puede ser accedida y su información podría perderse, tal como sucedía cuando explicamos la infraestructura anterior. Es por eso que una gran parte de la arquitectura está pensada al rededor de peers que opten por ser colaboradores, estos peers van a replicar todas las bases de datos que existan actualmente en la base de datos central, pensandolo en el caso del repositorio, van a preservar todos los articulos que existan. De esta manera logramos que mientras haya por lo menos un colaborador en linea, la wiki va a poder ser accedida.

Estos peers colaboradores son el punto más importante que diferencia esta solución con el resto y que sigue con la filosofía de las aplicaciones comunitarias que estamos analizando, permitiendo que la disponibilidad de la información se este logrando a través de la donación de almacenamiento en vez de dinero.

Entonces, un usuario que quiera conectarse a la base de datos y obtener su información, por ejemplo querer conectarse a la wiki y ver artículos, debe primero conectarse a alguno de estos

colaboradores, de los cuales puede replicar la información a su base de datos propia. Sin embargo esto no es tan trivial como parece, pasemos a ver como se resuelve este problema.

Manejo de conexión OrbitDB no se responsabiliza de manejar las conexiones, tampoco le importa, ya que te asegura que eventualmente la base de datos va a estar sincronizada entre peers, aunque se caigan las conexiones o te conectes mas tarde, todo se va a sincronizar sin conflictos. Por lo tanto delega esa responsabilidad a **Helia**[19] la cual es la implementación de IPFS en los lenguajes javascript/typescript, que a su vez delega la responsabilidad a **LibP2P**[38] y de la cual tenemos que hacer uso nosotros para manejar las conexiones.

LibP2P es una colección de protocolos y utilidades para facilitar la implementación de una red peer-to-peer. Al crear un nodo de LibP2P se tiene que elegir como conformarlo en base a un conjunto modular de herramientas, dentro de los que se encuentran mecanismos de seguridad, de transporte, descubrimiento de pares, entre otros. Cada una de estas herramientas es importante y hace que el nodo funcione como queramos, más adelante explicaremos la decisión para cada herramienta elegida, sin embargo ahora nos vamos a centrar en dos protocolos de interés necesarios para solucionar el problema de manejo de conexión. Los protocolos de transporte y de descubrimiento de peers.

Los 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. Como lo son TCP, QUIC o WebSockets. Sin embargo no todos los transportes nos son útiles y la decisión de cual protocolo utilizar no es sencilla.

Una de los requisitos de esta infraestructura era la posibilidad de utilizarla para crear aplicaciones que se puedan utilizar desde un entorno web, como es el caso de un usuario queriendo acceder a la wiki desde un navegador. Lo que sucede es que el entorno web no es muy amigable con las conexiones. Los navegadores están construidos sobre HTTP(S), un protocolo sin estado basado en solicitudes y respuestas. El cliente (navegador) envía una solicitud y luego espera una respuesta. Este modelo unidireccional y síncrono da como resultado una transferencia de datos lenta. Las conexiones se manejan en la capa de transporte y no mediante HTTP(S). Una conexión TCP subyacente es utilizada por HTTP/1.1 y HTTP/2, o una conexión QUIC en el caso de HTTP/3. Para mantener la seguridad de los usuarios, los navegadores imponen reglas estrictas, como los requisitos de certificados y el bloqueo de políticas de origen cruzado (cross-origin).

Es por esto que por razones de seguridad, no es posible que un navegador establezca una conexión TCP o QUIC sin procesar directamente desde el navegador, ya que todas las conexiones deben cumplir con los requisitos de Contexto Seguro (Secure Context), como por ejemplo que los mensajes se entreguen a través de TLS.

Es por esta razón que la infraestructura debe hacer una diferenciación entre nodos que se ejecuten desde un entorno independiente, como es el caso de **node.js**[43] y nodos que se ejecuten en un entorno web, únicamente para asignarle los protocolos de transporte correspondientes a su entorno.

Hay 2 conexiones que se tienen que resolver, la conexión entre nodos independientes y la conexión desde un nodo web a un nodo independiente.

Para los nodos independientes el protocolo utilizado es TCP, con el cual pueden comunicarse entre sí y con otros nodos independientes de la red. El problema viene que, como explicamos, TCP no nos sirve para comunicar desde un nodo web a un nodo independiente, por lo tanto hay que explorar alternativas.

Al momento de comenzar con el desarrollo del repositorio de conocimiento, la implementación de LibP2P en javascript soportaba

AstraDB Para entender cómo funciona AstraDB, primero entendamos resumidamente qué es.

AstraDB es una abstracción de una base de datos "key-value like.ºrientada para aplicaciones comunitarias en el ecosistema de IPFS. Es una base de datos descentralizada y distribuída, peer-to-peer, la cual hace uso de tecnologías como **OrbitDB** [45] para el manejo de datos y **LibP2P**

[38] para la comunicación peer-to-peer entre usuarios.

Su arquitectura tiene 2 grandes responsabilidades. Por un lado tiene que encargarse del manejo de lo datos, permitir que la aplicación tenga un estado y este pueda ser modificado por parte de cualquier usuario. Por otro lado debe proveer la forma de conectar y proveer de información a los distintos usuarios, para que estos puedan comunicarse y notificarse de nuevos cambios.

Manejo de los datos

8.4. Blockchain

En este trabajo se utilizó la red de Ethereum, al ser una blockchain popular nos permite demostrar y comparar los casos de uso contra nuestra solución en IPFS. Ethereum está compuesta de nodos distribuidos que comparten poder de cómputo lo cual permite el desarrollo de aplicaciones descentralizadas. Cuenta con una moneda 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.

8.4.1. Swarm

Para el desarrollo del sitio web estático se decidió ir por Swarm que es un almacenamiento descentralizado que corre sobre una *sidechain* de Ethereum. Swarm surgió como uno de los tres pilares de Ethereum para una web descentralizada [46]. Funciona por *content addressing* como IPFS e incluye un modelo de incentivos utilizando su propia moneda llamada BZZ.

Feed Los *feeds* en Swarm funcionan de manera similar a los IPNS de 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.

8.4.2. Ethereum

Para los casos de uso del repositorio de conocimiento y el mensajero en tiempo real necesitamos una herramienta que nos 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 15: *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).

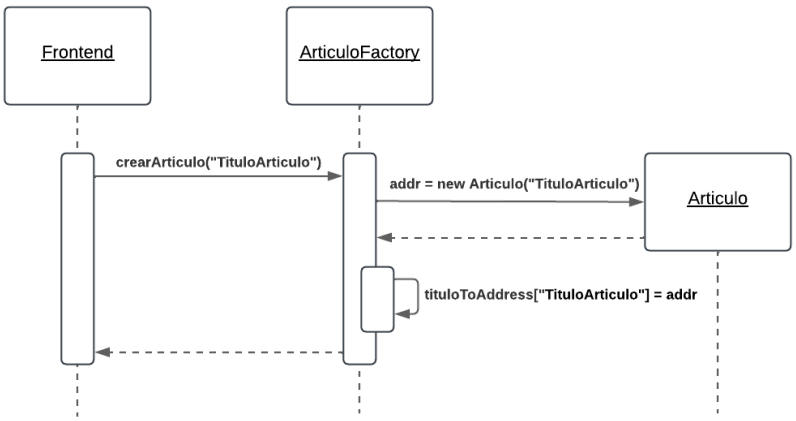

Figura 16: Creación de un artículo

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 17.

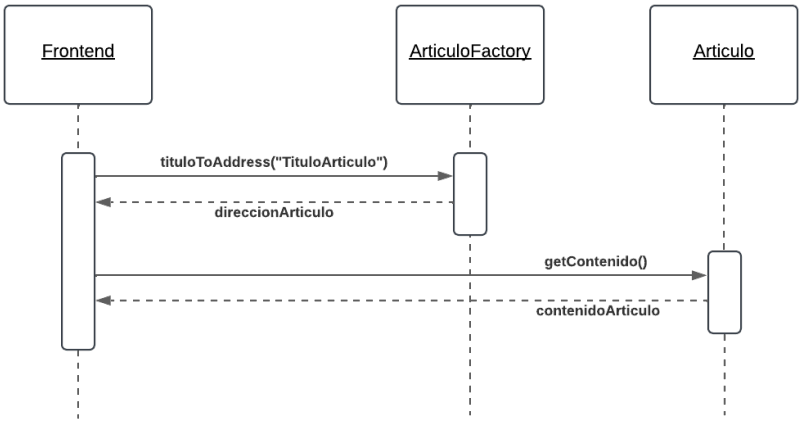

Figura 17: Obtención del contenido de un artículo

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 frontend. Al recibir este evento el frontend 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.

8.5. Front-end

Como prueba de la versatilidad de los ecosistemas, y aprovechando la creación de paquetes, se desarrollaron diferentes front-ends para las aplicaciones realizadas.

Astraweb El principal front-end que contiene el repositorio de conocimiento y a su vez el mensajero en tiempo real. Además, posee la opción de escoger el ecosistema al que se desea conectar.

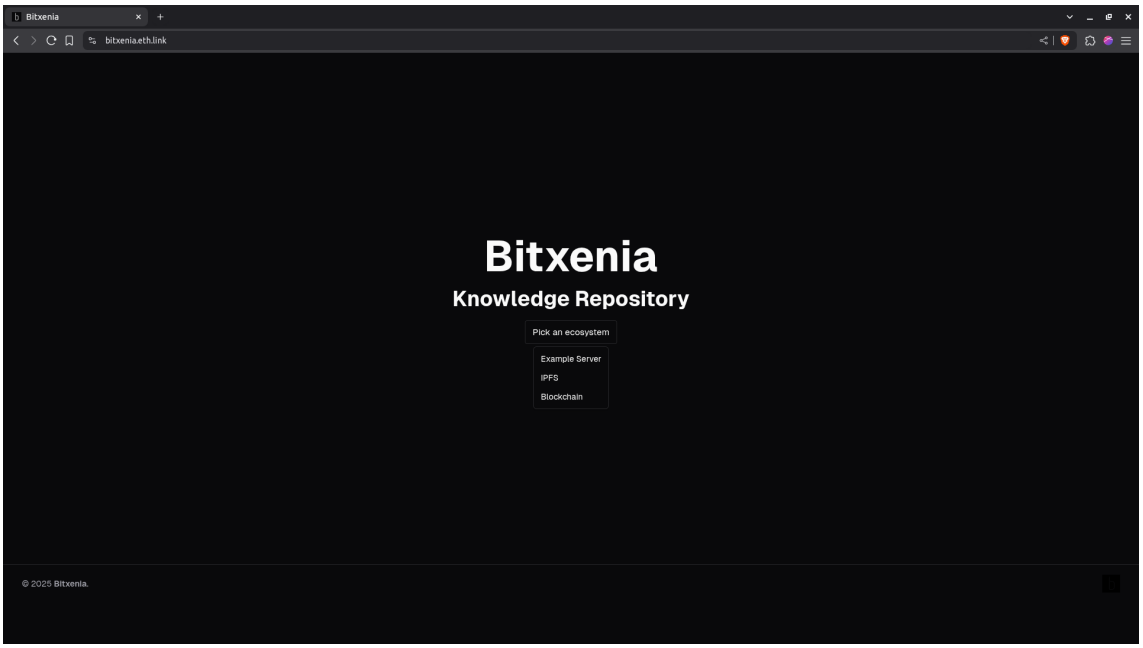

Figura 18: Página principal de Astraweb

Tecnología Se utilizó `React` [50] y `Next.js` [51] como *frameworks* para la creación de la aplicación web, basándonos en una plantilla llamada `rubix-documents` [53]. El código fue escrito en `Typescript`.

Servidor ejemplo Se desarrolló una solución centralizada como tercer ecosistema para este frontend, con dos propósitos:

1. Paralelizar el desarrollo del front-end 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.
2. Comparar las implementaciones descentralizadas contra un enfoque centralizado.

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 [54] 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` [9], 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, el cual se basa en diferentes contenedores y se verá más adelante.

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 19: 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 Blockchain.

Figura 20: Página principal de `astrachat-cli`

Tecnología Se utilizó `React` [50] y `Ink` [26] como *frameworks* para crear la interfaz de este frontend.

Limitaciones Debido a que se ejecuta en la terminal no es posible hacer uso de una wallet como Metamask [40] ya que la misma sólo funciona en web. Debido a esto, se *hardcodearon* algunas de las wallets que provee Hardhat [18].

9. Metodología

El desarrollo se dividió en sprints semanales para los cuales utilizamos un tablero Kanban en Github Projects donde fuimos agregando las tareas a realizar para cada caso de uso. 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.

La modalidad fue en su mayoría virtual y asincrónica (excepto por la reunión semanal antes mencionada en la cual los integrantes del trabajo nos reunimos sincrónicamente). Nos mantuvimos en constante comunicación a través de un servidor de Discord y, también se realizaron sesiones de *pair* y *mob-programming* en distintas ocaciones.

10. Experimentación y/o validación

En base a los casos de uso implementados analizamos cada ecosistema en las siguientes categorías: costos, experiencia de desarrollo, viabilidad y performance, con el objetivo de compararlas y ver las ventajas y desventajas de cada uno.

10.1. Costos

¿Cuánto nos cuesta desplegar y mantener un servicio en cada ecosistema?

10.1.1. IPFS

10.1.2. Blockchain

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. Hay que tener en cuenta que 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.

¡TODO: medir cuánto es el costo aproximado en USD o BZZ¿

Ethereum Se utiliza la moneda ETH para pagar por el despliegue de cada transacción, esto incluye tanto el despliegue de cada *smart contract* como también la edición de un artículo (en el caso del repositorio de conocimiento). Por lo tanto, el usuario final de la aplicación termina pagando por 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.

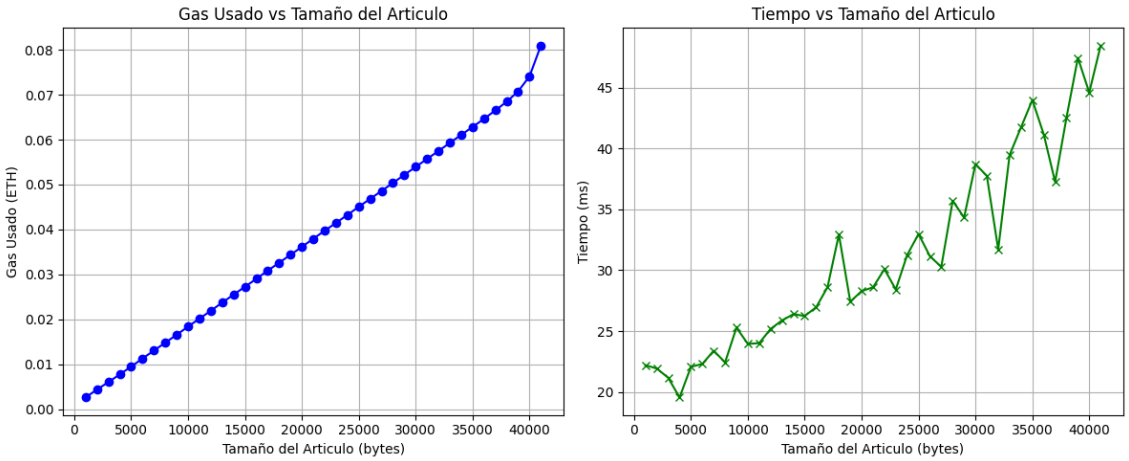

Figura 21: Tiempo y gas usado al crear artículos con tamaño de bytes creciente

10.2. Experiencia de desarrollo

¿Qué tan fácil es desplegar en cada ecosistema?

10.2.1. IPFS

10.2.2. Blockchain

Swarm En Swarm existe la herramienta de terminal swarm-cli con la cual se puede interactuar con un nodo de Swarm. 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.

Ethereum 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.

10.3. Viabilidad

¿Que tan viable es crear una aplicación comunitaria para cada uno de estos ecosistemas?

10.3.1. IPFS

10.3.2. Blockchain

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.

10.4. Performance

10.4.1. IPFS

10.4.2. Blockchain

Astrachat-eth Las siguientes métricas corresponden a Astrachat. Se obtuvieron levantando una instancia local de Hardhat [18] y ejecutando 1000 muestras. A partir de los datos obtenidos se consiguieron el máximo (**Max**), mínimo (**Min**), media (**Mean**), desvío estándar (**Std**) y la mediana (**Median**).

Tiempo en enviar un mensaje La primer métrica tomada corresponde al tiempo que tarda en enviarse un mensaje corto (*Lorem ipsum*).

Max	11.69 ms
Mean	6.15 ms
Min	5.05 ms
Std	0.80 ms
Median	6.04 ms

Cuadro 1: Tiempo en enviar un mensaje

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 1 de junio de 2025 a las 22:05 hs de $2536.14 de la página Coinmarketcap.

	ETH	USD
Max	0.00049158	1.25
Mean	0.00034321	0.87
Min	0.00034239	0.87
Std	0.00000715	0.02
Median	0.00034239	0.87

Cuadro 2: Gas usado para enviar un mensaje

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 1000 mensajes). El gráfico muestra como el tiempo (en milisegundos) se va incrementando a medida que hay más mensajes en el chat.

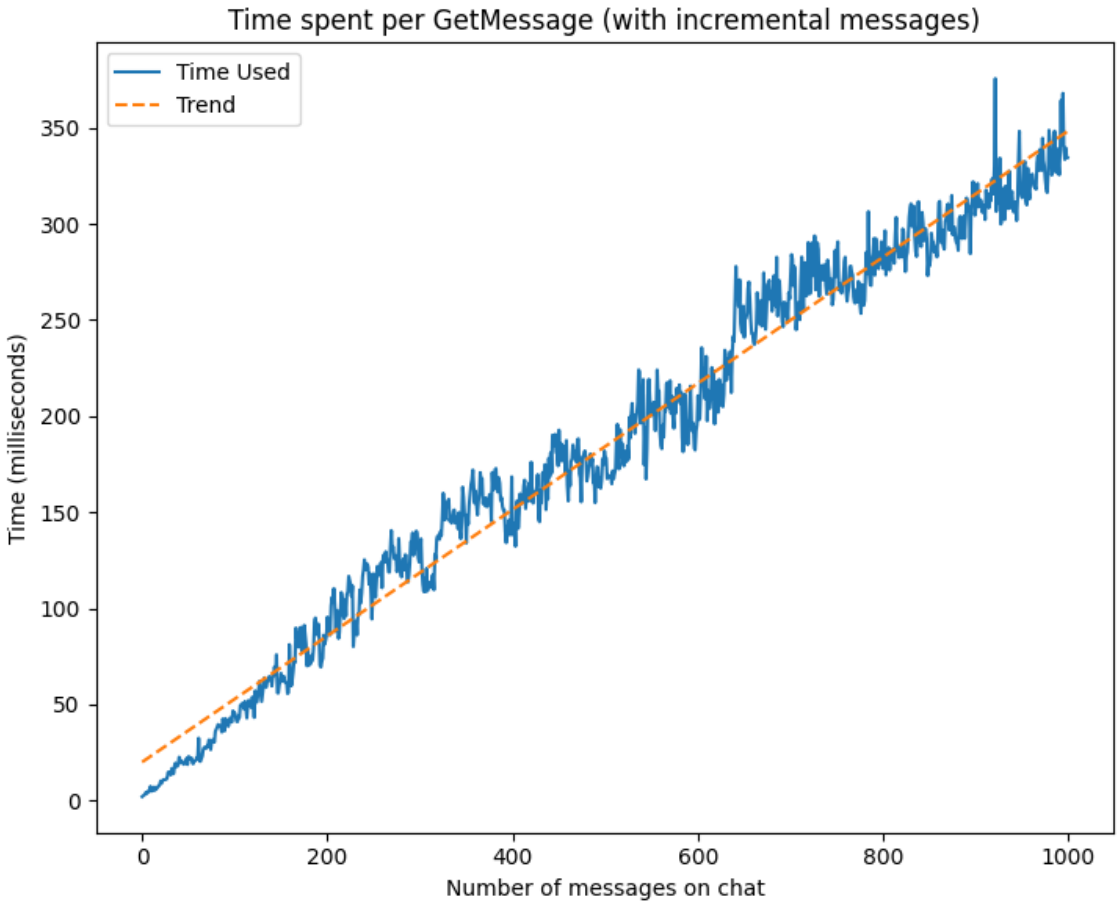

Figura 22: Tiempo para obtener mensajes de un chat según el tamaño del chat

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	9.70 ms
Mean	5.51 ms
Min	4.37 ms
Std	1.14 ms
Median	4.79 ms

Cuadro 3: 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	13.46 ms
Mean	5.75 ms
Min	4.68 ms
Std	0.93 ms
Median	5.27 ms

Cuadro 4: Tiempo en enviar y recibir un mensaje (distintos usuarios)

10.5. Resumen

	IPFS	**Blockchain**
Costos	Bajos o nulos	Escala con el uso de la aplicación
Desarrollo		Existen herramientas que facilitan el desarrollo y el despliegue
Viabilidad		Sitios estáticos y aplicaciones CRUD
Performance		

Cuadro 5: Comparación entre los ecosistemas de IPFS y Blockchain

11. Cronograma

Realizamos un cronograma tentativo de la totalidad del trabajo, incluyendo el desarrollo de cada caso de uso, el despliegue en cada ecosistema y su documentación asociada.

Cada caso de uso incluye una etapa de *Discovery* en la cuál definiremos su alcance y lo desglosaremos en tareas más concretas.

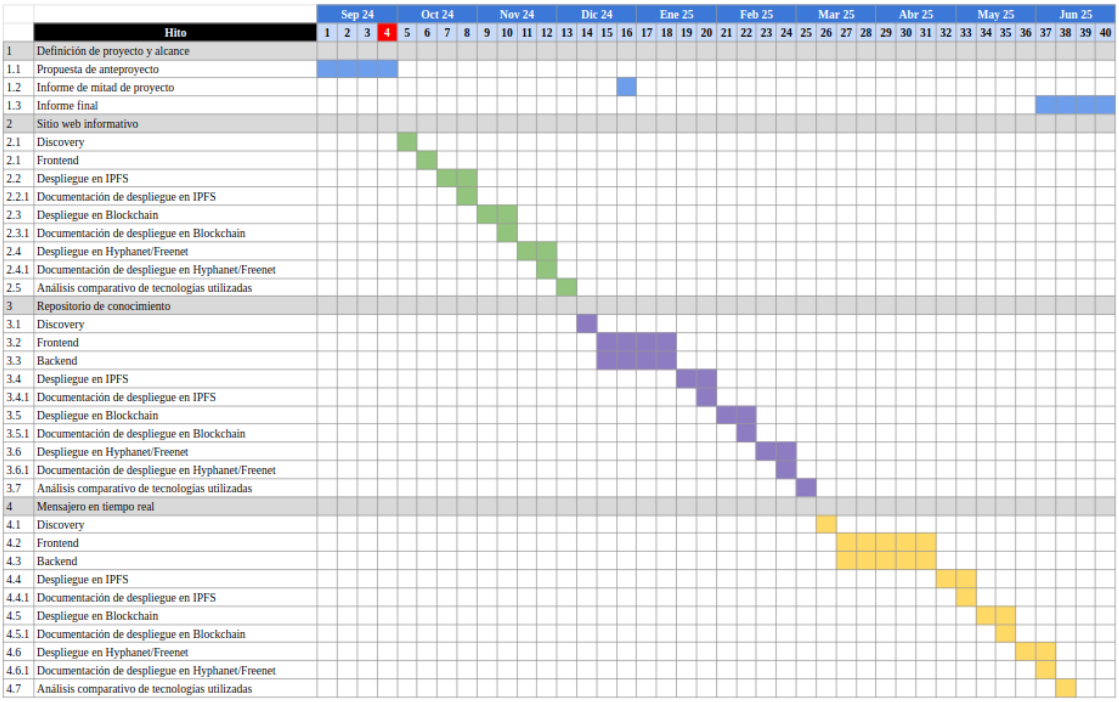

Figura 23: Cronograma tentativo

1. **Definición de proyecto y alcance**
   1.1. Propuesta de anteproyecto (Semanas 1 a 4)
   1.2. Informe de mitad de proyecto (Semanas 16 a 17)
   1.3. Informe final (Semanas 37 a 40)
2. **Sitio web informativo** (Semanas 5 a 13)
   2.1. Discovery
   2.2. Frontend
   2.3. Despliegue en IPFS
      2.3.1. Documentación de despliegue en IPFS
   2.4. Despliegue en Blockchain
      2.4.1. Documentación de despliegue en Blockchain
   2.5. Despliegue en Hyphanet/Freenet
      2.5.1. Documentación de despliegue en Hyphanet/Freenet
   2.6. Análisis comparativo de tecnologías utilizadas
3. **Repositorio de conocimiento** (Semanas 14 a 25)
   3.1. Discovery
   3.2. Frontend
   3.3. Backend
   3.4. Despliegue en IPFS
      3.4.1. Documentación de despliegue en IPFS
   3.5. Despliegue en Blockchain
      3.5.1. Documentación de despliegue en Blockchain
   3.6. Despliegue en Hyphanet/Freenet
      3.6.1. Documentación de despliegue en Hyphanet/Freenet
   3.7. Análisis comparativo de tecnologías utilizadas
4. **Mensajero en tiempo real** (Semanas 26 a 38)
   4.1. Discovery
   4.2. Frontend
   4.3. Backend
   4.4. Despliegue en IPFS
      4.4.1. Documentación de despliegue en IPFS
   4.5. Despliegue en Blockchain
      4.5.1. Documentación de despliegue en Blockchain
   4.6. Despliegue en Hyphanet/Freenet
      4.6.1. Documentación de despliegue en Hyphanet/Freenet
   4.7. Análisis comparativo de tecnologías utilizadas

12. 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.

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

Cuadro 6: Riesgos materializados

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.

13. Lecciones aprendidas

- 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.
- Al trabajar con distintos ecosistemas, modularizar en distintos paquetes/librerías cada uno facilita la integración, las pruebas y el cálculo de métricas.
- Nos abrió al mundo P2P y web descentralizada, que es una manera distinta de pensar y desarrollar aplicaciones. [TODO LS: desarrollar]

14. Impactos sociales y ambientales

15. Trabajos futuros

16. Conclusiones

16.1. Conclusión del análisis

16.2. Conclusión general

17. Referencias

[1] *Anatomy of an IPNS name.* (s.f.). https://docs.ipfs.tech/concepts/ipns/#anatomy-of-an-ipns-name

[2] 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/
[3] Bondy, B. (2024). IPFS Support in Brave. *Brave Blog*. https://brave.com/blog/ipfs-support/
[4] Brnakova, J. (s.f.). *Game decommissioning: When beloved games shut down*. https://www.revolgy.com/insights/blog/game-decommissioning-when-beloved-games-get-shut-down-and-online-worlds-disappear
[5] *Censorship of Wikipedia*. (s.f.). https://en.wikipedia.org/wiki/Censorship_of_Wikipedia
[6] *Collaborative Clusters*. (s.f.). https://ipfscluster.io/documentation/collaborative/
[7] *Collaborative clusters setup*. (s.f.). https://ipfscluster.io/documentation/collaborative/setup/
[8] *Configuration Reference*. (s.f.). https://ipfscluster.io/documentation/reference/configuration
[9] *Daemon (computing)*. (s.f.). https://en.wikipedia.org/wiki/Daemon_(computing)
[10] *Data Differencing*. (s.f.). https://en.wikipedia.org/wiki/Data_differencing
[11] *Distributed Hash Tables (DHTs)*. (s.f.). https://docs.ipfs.tech/concepts/dht
[12] *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
[13] *DNSLink*. (s.f.). https://docs.ipfs.tech/concepts/dnslink/#dnslink
[14] *ENS*. (s.f.). https://ens.domains/
[15] *Fees*. (s.f.). https://support.ens.domains/en/articles/7900605-fees
[16] *Fleek*. (s.f.). https://fleek.xyz/docs/platform/hosting/
[17] *Freenet*. (s.f.). https://freenet.org
[18] *Hardhat*. (s.f.). https://hardhat.org/
[19] *Helia: IPFS node implementation in Javascript*. (s.f.). https://github.com/ipfs/helia
[20] *Host a single-page website with IPFS Desktop*. (s.f.). https://docs.ipfs.tech/how-to/websites-on-ipfs/single-page-website/#set-up-a-domain
[21] *How to facilitate sharding on collaborative clusters?* (s.f.). https://discuss.ipfs.tech/t/how-to-facilitate-sharding-on-collaborative-clusters/18052
[22] *HTTP API for IPFS Cluster*. (s.f.). https://ipfscluster.io/documentation/reference/api/
[23] *HTTP API for Kubo*. (s.f.). https://docs.ipfs.tech/reference/kubo/rpc/
[24] Hurtado, J. S. (s.f.). *Qué es Blockchain y cómo funciona la tecnología Blockchain*. Consultado el 25 de septiembre de 2024, desde https://www.iebschool.com/blog/blockchain-cadena-bloques-revoluciona-sector-financiero-finanzas
[25] *Hyphanet*. (s.f.). https://www.hyphanet.org/index.html
[26] *Ink*. (s.f.). https://github.com/vadimdemedes/ink
[27] *InterPlanetary Name System*. (s.f.). https://docs.ipfs.tech/concepts/ipns
[28] *IPFS*. (s.f.). https://ipfs.tech
[29] *IPFS Cluster - Collaborative Clusters*. (s.f.). https://collab.ipfscluster.io/#list-of-clusters
[30] *IPFS Content Identifiers*. (s.f.). https://docs.ipfs.tech/concepts/content-addressing/#what-is-a-cid
[31] *IPFS Garbage Collector*. (s.f.). https://docs.ipfs.tech/concepts/persistence/#garbage-collection
[32] *IPFS Gateway*. (s.f.). https://docs.ipfs.tech/concepts/ipfs-gateway/
[33] *IPFS Pinning services*. (s.f.). https://docs.ipfs.tech/concepts/persistence/#pinning-services
[34] *ipfs-cluster-follow*. (s.f.). https://ipfscluster.io/documentation/reference/follow/
[35] *IPNS Record and Protocol*. (s.f.). https://specs.ipfs.tech/ipns/ipns-record/
[36] Janardhan, S. (s.f.). *More details about the October 4 outage*. https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/
[37] *Kubo: IPFS node implementation in Go*. (s.f.). https://docs.ipfs.tech/install/command-line/
[38] *libp2p*. (s.f.). https://libp2p.io/
[39] *Markdown*. (s.f.). https://en.wikipedia.org/wiki/Markdown
[40] *Metamask*. (s.f.). https://metamask.io/
[41] 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
[42] *Multiaddr*. (s.f.). https://multiformats.io/multiaddr/

[43] *Node.js*. (s.f.). https://nodejs.org/es
[44] *An O(ND) Difference Algorithm and Its Variations*. (s.f.). https://neil.fraser.name/writing/diff/myers.pdf
[45] *OrbitDB*. (s.f.). https://orbitdb.org
[46] *The Origins of Swarm*. (2015). https://blog.ethswarm.org/hive/2024/the-origins-of-swarm
[47] *Pinata*. (s.f.). https://pinata.cloud/ipfs
[48] *Pinning*. (s.f.). https://docs.ipfs.tech/concepts/glossary/#pinning
[49] *Public Gateway Checker*. (s.f.). https://ipfs.github.io/public-gateway-checker/
[50] *React*. (s.f.). https://react.dev/
[51] *The React Framework for the Web*. (s.f.). https://nextjs.org/
[52] *Repositorio del nodo confiable*. (s.f.). https://github.com/bitxenia/astrawiki-web-trusted-peer
[53] *rubix-documents*. (s.f.). https://github.com/rubixvi/rubix-documents
[54] *Server Components*. (s.f.). https://react.dev/reference/rsc/server-components
[55] *A Universally Unique IDentifier (UUID) URN Namespace*. (s.f.). https://www.rfc-editor.org/rfc/rfc4122

18. Anexos

18.1. Herramientas de LibP2P utilizadas