Universidad Austral de Chile

Facultad de Ciencias de la Ingeniería
Escuela de Ingeniería Civil en Informática

PROPUESTA DE APLICACIÓN DISTRIBUIDA PARA SINCRONIZACIÓN DE INFORMACIÓN EN AMBIENTES *EDGE COMPUTING*

Proyecto para optar al título de
Ingeniero Civil en Informática

PATROCINANTE:
JUAN ALEJANDRO SEBASTIÁN MARDONES YÁÑEZ
INGENIERO CIVIL EN INFORMÁTICA

PROFESOR CO-PATROCINANTE:
CRISTIAN ALEJANDRO OLIVARES RODRÍGUEZ
INGENIERO CIVIL INFORMÁTICO
MÁSTER EN VISIÓN POR COMPUTADORA E INTELIGENCIA ARTIFICIAL
DOCTOR EN INGENIERÍA

PROFESOR INFORMANTE
MAURICIO RUIZ-TAGLE MOLINA
INGENIERO CIVIL EN INFORMÁTICA
DOCTOR EN SOFTWARE Y SISTEMAS

DIEGO ALBERTO SANDOVAL BURGOS

VALDIVIA – CHILE
2023

ÍNDICE

ÍNDICE DE TABLAS

Tabla Página

ÍNDICE DE FIGURAS

Figura Página

RESUMEN

Este proyecto presenta una propuesta de aplicación distribuida basada en *blockchain* para sincronizar datos en la red de un proveedor de servicios de comunicaciones o CSP (*Communication Service Provider*).

La red está compuesta por nodos VSB (*Very Smart Box*), computadores de propósito general que contribuyen en la prestación del servicio de red, ubicados en las dependencias de los clientes del CSP; y por un nodo "autorizado" que envía información a los nodos VSB. La información enviada es confidencial entre el nodo autorizado y cada uno de los nodos VSB, de forma que otros nodos no pueden acceder a la información privada. Se utiliza una *blockchain* basada en Quorum, una variante de Ethereum que permite transacciones privadas mediante criptografía de clave pública.

La implementación del sistema consiste en un contrato inteligente desplegado en esta *blockchain*, así como la definición de programas o *scripts* que permiten a los nodos interactuar con dicho contrato. Se demuestra la factibilidad técnica de la propuesta al permitir la sincronización de datos en un entorno de pruebas. Además, se mejora la trazabilidad de las configuraciones, ya que la tecnología *blockchain* asegura la integridad e inmutabilidad de los datos, permitiendo a un operador autorizado revisar el historial de transacciones de cada nodo. Se explican los procesos necesarios para habilitar el sistema propuesto, incluyendo la instalación de software específico en los nodos VSB y el mantenimiento de una lista de correspondencia entre identificadores y claves públicas.

Este trabajo destaca las ventajas que ofrece el uso de *blockchain* para la sincronización de datos, abriendo nuevas oportunidades para mejorar la gestión y la seguridad en entornos de comunicación.

ABSTRACT

This project presents a proposal for a distributed application based on blockchain to synchronize data in a Communications Service Provider's (CSP) network.

The network consists of VSB (*Very Smart Box*) nodes, general purpose computers that support the network service provision and are located at the CSP's clients' premises; and an "authorized" node that sends information to the VSB nodes. The information sent is confidential between the authorized node and each VSB node, ensuring that other nodes cannot access private information. The proposal suggests the use of a blockchain based on Quorum, a variant of Ethereum that enables private transactions using public-key cryptography.

The system implementation consists of a smart contract deployed on this blockchain, as well as the definition of programs or scripts that allow the nodes to interact with this contract. The technical feasibility of the proposal is demonstrated by enabling data synchronization in a testing environment. Furthermore, the traceability of configurations is improved as blockchain technology ensures data integrity and immutability, allowing an authorized operator to review the transaction history of each node. The necessary processes to enable the proposed system are explained, including the installation of specific software on the VSB nodes and the maintenance of a correspondence list between identifiers and public keys.

This work highlights the advantages offered using blockchain for data synchronization, opening new opportunities to enhance management and security in communication environments.

1. INTRODUCCIÓN

1.1 Contexto

1.1.1 Grupo GTD

Grupo GTD es una empresa chilena proveedora de servicios de comunicaciones (CSP[1]). Fue fundada en 1979 y ofrece servicios de telefonía fija, telefonía móvil, acceso a Internet, así como servicios de TI[2] y soluciones en la nube para empresas (Grupo GTD, 2023a).

La compañía tiene presencia en seis países: Chile, Colombia, Perú, Ecuador, Italia y España (Grupo GTD, 2023b). Se destaca por ser pionera en tecnología y por proporcionar a sus clientes "soluciones tecnológicas de vanguardia ajustada a sus necesidades" (Grupo GTD, 2023c).

Además, Grupo GTD se caracteriza por su constante inversión en infraestructura tecnológica. En 2021, en colaboración con el Gobierno de Chile, inauguró el cable submarino Prat, con una longitud de 3500 kilómetros, que abarca desde Arica hasta Puerto Montt, pasando por Iquique, Tocopilla, Antofagasta, Caldera, La Serena, Concón, Cartagena, Constitución y Puerto Saavedra (Bertolini, 2021). Este proyecto ha conectado territorios históricamente aislados, facilitando la transformación digital de las empresas locales y mejorando la experiencia de los clientes, al reducir la probabilidad de fallos (Grupo GTD, 2022).

Adicionalmente, la empresa cuenta con una red internacional de centros de datos (*datacenter*) que ha experimentado un crecimiento en los últimos años. Estos centros de datos poseen certificación *Tier III*, lo que garantiza un tiempo de actividad (*uptime*) anual de al menos el 99.98% (Grupo GTD, 2023c).

En 2023, Grupo GTD se convierte en el distribuidor autorizado de Starlink en Chile, Perú y Colombia. Esta asociación permite a la compañía contar con una ruta de respaldo para sus clientes y brindar servicio en áreas remotas (Grupo GTD, 2023d).

Actualmente, la empresa cuenta con más de 285 mil clientes residenciales y más de 35 mil clientes de empresas y corporaciones (Grupo GTD, 2023e).

---

[1] CSP: *Communication Service Provider*

[2] TI: Tecnologías de la Información

1.1.2 Equipamiento de red

Las empresas proveedoras de servicios de comunicaciones (CSP) despliegan equipamiento de red en las dependencias de sus clientes cuando el servicio así lo requiere. Este equipamiento incluye, entre otros: enrutadores, puntos de acceso de red inalámbrica (*Access Point*), terminales de red óptica (ONT[3]), cableado y, en algunos casos, computadores de propósito general programados para cumplir funciones de red.

En el caso de Grupo GTD, se instalan computadores de propósito general en las instalaciones de sus clientes del segmento empresial. Estos computadores, conocidos internamente como VSB (*Very Smart Box*), desempeñan una serie de tareas operativas y de apoyo al servicio, que incluyen:

1. **Servidor DHCP[4]:** Entrega direcciones IP[5] de forma automática a los equipos que se conectan a la red del cliente.

2. **Servidor DNS[6]:** Resuelve consultas de nombres de dominio. Es decir, traduce de nombres de dominio a direcciones IP para los equipos que se conectan a la red del cliente.

3. **Servidor SIP[7]:** Realizar y recibir llamadas mediante telefonía IP (VoIP[8]).

4. **Servidor TFTP[9]:** Usado para transferir archivos con metadatos del servicio hacia los servidores de la compañía. Entre los datos enviados se encuentran: registros de llamadas, grabaciones de voz para contestador automático (para llamadas en espera), actualizaciones de firmware de los dispositivos VoIP.

El funcionamiento de los VSB depende de archivos de configuración almacenados localmente en su disco duro. Sin embargo, la instalación de estos archivos se realiza manualmente, lo que puede dar lugar a errores. Además, no se mantiene un registro, ni local ni remoto, del historial de configuración de cada VSB. Esto implica que la

---

[3] ONT: *Optical Network Terminal*

[4] DHCP: *Dynamic Host Configuration Protocol*

[5] IP: *Internet Protocol*

[6] DNS: *Domain Name System*

[7] SIP: *Session Initiation Protocol*

[8] VoIP: *Voice over IP*

[9] TFTP: *Trivial File Transfer Protocol*

configuración local es volátil y no se puede rastrear fácilmente. La falta de trazabilidad dificulta el diagnóstico de problemas y la respuesta a fallos operacionales y de ciberseguridad.

La solución más común y básica para abordar este problema consiste en utilizar uno o varios servidores ubicados en las instalaciones del proveedor de servicios de comunicaciones (CSP) que almacenen los archivos de configuración y su historial, distribuyéndolos a los computadores VSB correspondientes.

En la Figura 1 se muestra esta configuración. El servidor centralizado se conecta directamente a cada VSB cuando necesita enviar la configuración, y cada VSB solo se comunica con el servidor centralizado, no con los demás dispositivos VSB en la red.

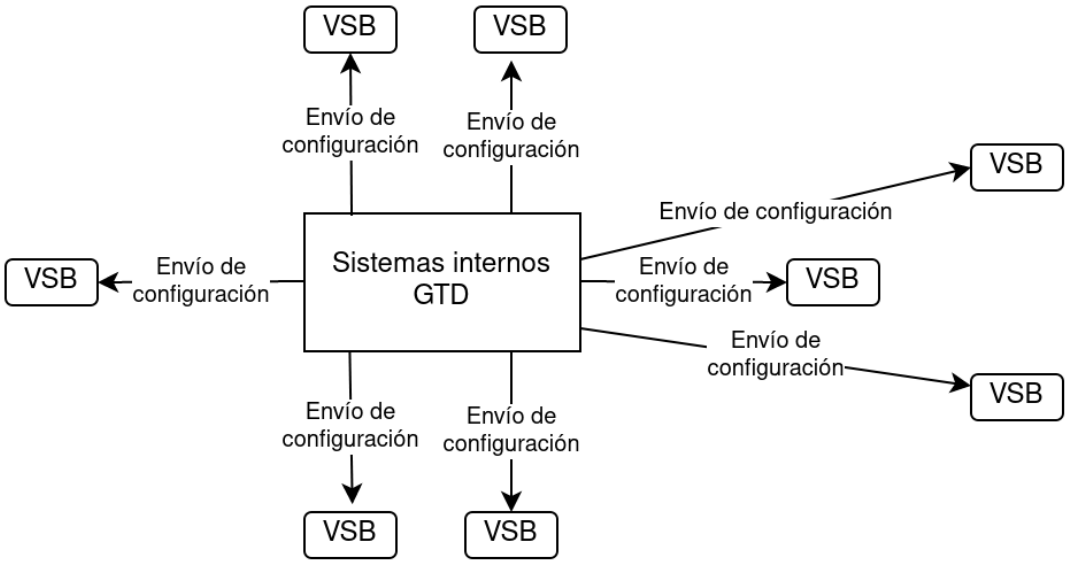

Figura 1: Sistema centralizado de distribución de configuraciones

En la presente investigación, se evalúa una alternativa descentralizada como solución al problema descrito.

1.2 Objetivos

1.2.1 Objetivo general

Evaluar una arquitectura distribuida que permita la sincronización de información entre nodos que formen parte de un ambiente *Edge Computing*.

1.2.2 Objetivos específicos

1. Diseñar un sistema de sincronización que mediante una red distribuida permita a los VSB obtener configuraciones al estar en la red del cliente.

2. Automatizar el proceso de sincronización de configuración mediante software que se ejecuta en el VSB.

3. Mejorar la persistencia de las configuraciones mediante una arquitectura distribuida.

4. Evaluar el funcionamiento del sistema y su aplicabilidad en el contexto de un CSP.

2. MARCO TEÓRICO

2.1 Sistemas distribuidos

Un sistema distribuido es un sistema informático en el que múltiples componentes o nodos interconectados colaboran para cumplir una tarea o proporcionar un servicio (Tanenbaum & van Steen, 1996). A diferencia de los sistemas centralizados, los sistemas distribuidos distribuyen la carga de trabajo, los recursos y la información entre los nodos, lo que los hace más flexibles (Tanenbaum & van Steen, 1996) y resistentes a fallos (Ledmi, Bendjenna, & Hemam, 2018).

Los nodos pueden ser computadoras independientes, servidores, dispositivos móviles o sensores, y se comunican y coordinan a través de redes de comunicación como redes de área local (LAN), redes de área amplia (WAN) o Internet.

Una de las aplicaciones computacionales más populares al comienzo de este siglo, y que se popularizó gracias a la masificación de Internet, fue el protocolo BitTorrent (Pasick, 2004). BitTorrent se ejecuta sobre una red *peer-to-peer* (o red de pares) donde diversos nodos alrededor del mundo comparten archivos sin que estos deban estar almacenados en un servidor centralizado (Cohen, 2002).

Durante las últimas dos décadas, han surgido diversos protocolos, productos y aplicaciones que también hacen uso de recursos computacionales distribuidos. Notables son proyectos como IPFS que buscan preservar información de hipertexto (sitios web) en una red distribuida resiliente que no dependa de una entidad central (Benet, 2019).

Además, desde la popularización de Bitcoin (Nakamoto, 2008) en la década pasada, la *blockchain* como estructura de datos ha tomado una gran relevancia en el campo de la informática (Chen, Xu, Shi, Zhao, & Zhao, 2019), y ha habido en los últimos años un creciente interés por su uso en sistemas *peer-to-peer* en ámbitos que no necesariamente tienen que ver con Bitcoin y criptomonedas, como lo puede ser el almacenamiento de datos de dispositivos IoT (*Internet of Things*) (Shafagh; Burkhalter; Hithnawi; Duquennoy, 2017).

A continuación, se exponen los avances más importantes en el ámbito de los sistemas distribuidos y, más específicamente, en la tecnología *blockchain* durante las últimas dos décadas.

2.2 Bitcoin

Bitcoin es un sistema de transacciones electrónicas descentralizadas, y un sistema de dinero digital, donde una red de pares (*peer-to-peer)* es capaz de llegar a un consenso

global sobre el historial de transacciones del sistema sin necesidad de una autoridad central (Nakamoto, 2008).

En los sistemas de pago electrónico tradicionales, como las tarjetas de crédito o las transferencias bancarias, se necesita un intermediario centralizado, como un banco, para validar las transacciones y actualizar el estado de cuenta de los participantes involucrados.

Bitcoin, sin embargo, logra resolver el problema del doble gasto, o *double-spending problem*, sin necesidad de una entidad centralizada (Nakamoto, 2008).

El doble gasto se refiere a la posibilidad de gastar la misma cantidad de dinero digital dos veces, lo cual era uno de los puntos débiles de los sistemas de dinero digital descentralizado propuestos hasta ese momento: si los nodos participantes de la red no lograban un consenso sobre cuáles eran las transacciones válidas, un participante malicioso podría aprovechar esta falta de consenso para engañar a los demás participantes y gastar dinero digital que ya había sido gastado previamente.

Bitcoin soluciona este problema mediante el uso de dos tecnologías: *blockchain* (cadena de bloques) y el mecanismo de consenso *Proof of Work* (prueba de trabajo) (Nakamoto, 2008).

2.2.1 ***Blockchain* y la función *hash***

La *blockchain*, o cadena de bloques, es una estructura de datos compuesta por bloques de información ordenados secuencialmente. En Bitcoin, es utilizada como un registro público y distribuido en el que se ingresan todas las transacciones realizadas. Conforme se realizan transacciones en el sistema, se van agregando más bloques (Brakeville & Perepa, 2019).

Cada bloque es identificado con el resultado de una función hash, que recibe como entrada las transacciones del bloque y el hash identificador del bloque anterior (Brakeville & Perepa, 2019), tal como se muestra en la Figura 2.

La función hash es un componente esencial de la cadena de bloques. Es un algoritmo criptográfico que toma una entrada (como un bloque de transacciones) y genera una salida única y de longitud fija llamada hash. El hash actúa como una huella digital única para cada bloque, y cualquier cambio en los datos de entrada produce un hash completamente diferente (Brakeville & Perepa, 2019).

Dado que cada bloque incluye el hash del bloque anterior, los bloques quedan enlazados, y se asegura la integridad y transparencia de las transacciones, ya que cualquier intento de modificar las transacciones de un determinado bloque requeriría cambiar todos los bloques siguientes (Brakeville & Perepa, 2019).

Si alguien intentara cambiar todos los bloques siguientes al bloque manipulado, esto resultaría extremadamente difícil, porque se necesitaría validar cada uno de esos bloques nuevos de la cadena manipulada mediante el mecanismo Proof of Work antes de que el resto de la red valide nuevos bloques en la cadena sin manipular (Buterin, 2014).

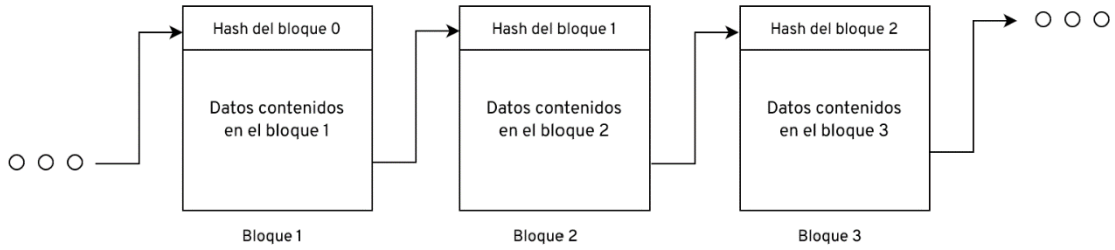

Figura 2: Representación de la *blockchain*, elaboración propia

2.2.2 **Mecanismo *Proof of work***

El mecanismo *Proof of Work*, ideado por Dwork y Naor (1992), Adam Back (2002), y luego adaptado por Satoshi Nakamoto (2008) para validar y confirmar las transacciones en Bitcoin, consiste en que ciertos nodos de la red, llamados mineros, compiten entre sí para resolver problemas criptográficos complejos y agregar nuevos bloques con nuevas transacciones a la cadena de bloques. El minero que resuelve el problema primero tiene el derecho de agregar un nuevo bloque a la cadena de bloques y es recompensado con bitcoins.

En caso de generarse varias cadenas paralelas como resultado de un intento de fraude, se considera que la cadena más larga es la válida, ya que implica un mayor trabajo o costo computacional (Nakamoto, 2008). De este modo, para poder instaurar como legítima una cadena adulterada, se requeriría una cantidad masiva de poder computacional. Más específicamente, el actor malicioso necesitaría tener más poder computacional que todo el resto de la red en su conjunto (Chohan, 2021). Esta situación hipotética se conoce como un ataque de 51%, o *51% attack* (Buterin, 2014). Aponte-Novoa, Orozco, Villanueva-Polanco et al (2021) plantean un análisis detallado al respecto.

De esta forma, Bitcoin garantiza la seguridad y la integridad de las transacciones, incluso en un entorno donde algunos participantes puedan ser deshonestos o adversarios.

2.3 Ethereum: contratos inteligentes y aplicaciones descentralizadas

En 2015 se lanzó Ethereum, una plataforma descentralizada de *blockchain* que ofrece funcionalidades adicionales a las de una criptomoneda convencional como Bitcoin (Buterin, 2014). Además de ser una red para transferir valor a través de su moneda digital,

llamada Ether (ETH), Ethereum permite la ejecución de contratos inteligentes y la creación de aplicaciones descentralizadas (DApps) (Smith, 2023a).

Los contratos inteligentes son programas que se ejecutan en la *blockchain* de Ethereum. Estos contratos contienen reglas y lógica programada, lo que les permite facilitar, verificar y hacer cumplir la negociación o ejecución de acuerdos sin necesidad de intermediarios (Wackerow, 2022a).

La ejecución de los contratos inteligentes se lleva a cabo en la EVM (*Ethereum Virtual Machine*), una máquina virtual Turing completa presente en cada nodo de la red (Buterin, 2014). Cuando un nodo recibe una transacción que incluye una solicitud para ejecutar un contrato inteligente, verifica su validez y la ejecuta en su propia instancia local de la EVM. La EVM ejecuta el código del contrato paso a paso, siguiendo las instrucciones definidas en él. Esto puede implicar la modificación del estado de la *blockchain*, como la actualización de saldos de cuentas o la modificación de variables en el contrato (Wood, 2023).

Cada nodo en la red posee una copia completa del estado de la *blockchain* de Ethereum, que incluye información sobre cuentas y contratos inteligentes. El estado se actualiza a medida que se ejecutan transacciones y contratos inteligentes. Cada cambio en el estado se registra en un nuevo bloque de la cadena, asegurando la consistencia en todos los nodos (Wood, 2023).

Una vez que un contrato inteligente está desplegado en la red, cualquier nodo puede interactuar con él enviando transacciones que invoquen a las funciones definidas en el contrato. Estas funciones pueden realizar cálculos, actualizar el estado, emitir eventos y responder a solicitudes de otros contratos o usuarios de la red (Wackerow, 2022a).

Ethereum utiliza su propia criptomoneda, Ether, para efectuar transacciones en la red y pagar por la ejecución de contratos inteligentes (Smith, 2023b).

El mecanismo de consenso de Ethereum inicialmente era *Proof of Work* (Buterin, 2014), pero en 2022 se completó la transición de Ethereum al mecanismo *Proof of Stake* (Prueba de Participación), un mecanismo alternativo que reduce el gasto energético de la red (Ethereum.org, 2023).

La capacidad de ejecutar contratos inteligentes en Ethereum ha abierto la puerta a una amplia gama de aplicaciones descentralizadas, conocidas como DApps. Estas aplicaciones se ejecutan en la *blockchain* y eliminan la necesidad de una autoridad centralizada. Las DApps pueden abarcar una variedad de casos de uso, como sistemas de votación, mercados descentralizados, juegos en línea y más (Crypto Council for Innovation, 2022).

En la Figura 3, elaborada por Boetticher (2023), se presenta una conceptualización de una DApp. En este escenario, el usuario interactúa con un *frontend*, como por ejemplo, una interfaz gráfica de usuario en una página web. El *frontend*, a su vez, interactúa con un *provider*, que es un cliente Ethereum que se ejecuta de forma remota, sin control directo del usuario. El *provider* expone una API que permite al *frontend* interactuar de manera indirecta y simplificada con el contrato inteligente.

Cuando el *Provider* interactúa con el contrato inteligente, debe ejecutar esta interacción en la EVM local, para luego distribuir los cambios a la *blockchain*. Si la transacción es válida, el resto de los nodos ejecutan el código correspondiente y agregan la transacción a la *blockchain*.

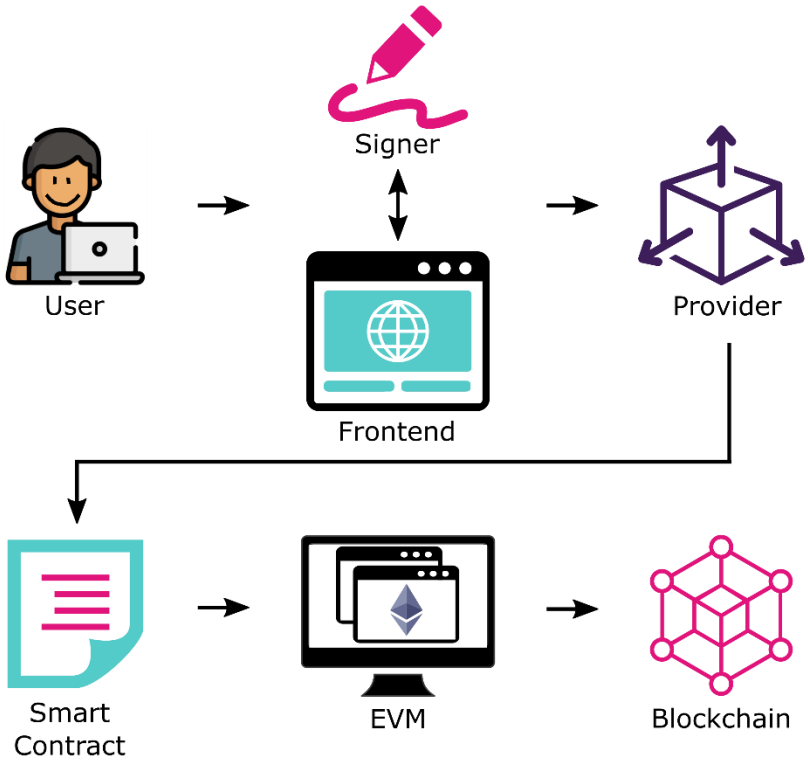

Figura 3: Componentes de una Dapp[10]

Para desarrollar contratos inteligentes en Ethereum, se utiliza el lenguaje de programación Solidity.

[10] Fuente: Boetticher, 2023

Solidity es un lenguaje de programación de alto nivel diseñado específicamente para el desarrollo de contratos inteligentes en plataformas *blockchain*, como Ethereum (Wackerow, 2022b). Su sintaxis está fuertemente influenciada por lenguajes como C++ (The Solidity Authors, 2023), lo cual facilita su adopción y curva de aprendizaje. Fue creado por Gavin Wood (Ethereum.org, 2023), quien es una de las figuras clave en el desarrollo de Ethereum.

Solidity permite a los desarrolladores definir la lógica del contrato, gestionar estados y transiciones, y especificar interacciones con otros contratos y usuarios de la red.

En la Figura 4, se detalla el código de un contrato muy simple en Solidity.

```
// Versión del compilador de Solidity
pragma solidity ^0.8.0;

// Declaración del contrato
contract SimpleStorage {
    // Variable de estado para almacenar la información
    string public data;

    // Función para guardar la información
    function setData(string memory _data) public {
        data = _data;
    }

    // Función para recuperar la información almacenada
    function getData() public view returns (string memory) {
        return data;
    }
}
```

Figura 4: Contrato simple en Solidity

Este contrato simple incluye una variable llamada 'data', una función llamada 'setData' para establecer un valor en esa variable, y otra función llamada 'getData' para obtener el valor almacenado.

Lo destacable de este contrato es que permite que una persona llame a la función desde su nodo en su computadora para asignar un valor a la variable. Una vez que los otros nodos validan la transacción, cualquier persona en cualquier parte del mundo que posea un nodo de Ethereum puede recuperar el valor asignado a la variable.

Este contrato solo permite almacenar información, pero hay contratos que permiten realizar cálculos e implementan lógica más avanzada.

De cierta forma, podría verse Ethereum como un “computador global” (Patron, 2016). Como es de esperar, interactuar con este computador global tiene un costo asociado. Cuando se realiza una transacción que modifica los datos almacenados en Ethereum o ejecuta un contrato inteligente, los usuarios deben pagar una tarifa conocida como "gas"

(Peaster, 2020). El costo de la transacción en gas depende de la complejidad de la operación y la cantidad de recursos computacionales requeridos. Esta tarifa de gas se paga en Ether (ETH). Los usuarios deben tener suficiente saldo de ETH en su cuenta para cubrir el costo de gas de sus transacciones.

Además de utilizar Ether para pagar el gas en transacciones de Ethereum, los participantes de la red tienen la capacidad de crear sus propios tokens. Estos tokens son activos digitales que se ejecutan en la *blockchain* de Ethereum y siguen ciertos estándares para garantizar su compatibilidad e interoperabilidad con otras aplicaciones y servicios (Crypto.com, 2022).

Uno de los estándares más conocidos para tokens en Ethereum es el Estándar de Token ERC-20. Este estándar define una serie de funciones y eventos que deben ser implementados por cualquier contrato inteligente que desee crear tokens compatibles con ERC-20. Al seguir este estándar, los tokens ERC-20 se vuelven intercambiables y pueden ser utilizados en una variedad de aplicaciones, como billeteras digitales, intercambios descentralizados y contratos inteligentes (Crypto.com, 2022).

Algunas de las funciones más comunes definidas por el estándar ERC-20 incluyen la posibilidad de consultar el saldo de un token, transferir tokens entre direcciones, aprobar el gasto de tokens en nombre de otro contrato o dirección y emitir eventos para notificar a los observadores sobre las acciones realizadas con los tokens.

Otro estándar popular es el Estándar de Token ERC-721, que se utiliza para la creación de tokens no fungibles (NFTs). A diferencia de los tokens ERC-20, los NFTs son únicos e indivisibles, lo que significa que cada token tiene un valor único y no se puede intercambiar de manera equivalente con otros tokens. Este estándar permite la creación y gestión de activos digitales únicos, como obras de arte digitales, coleccionables y tokens representativos de bienes físicos (Crypto.com, 2022).

Además de los estándares mencionados, existen otros estándares de token en Ethereum, como ERC-223, ERC-777 y ERC-1155, que ofrecen características adicionales y funcionalidades específicas (Crypto.com, 2022). Estos estándares se crearon para abordar ciertas limitaciones o mejorar aspectos particulares de los tokens en Ethereum, como la eficiencia en el manejo de tokens, la seguridad y la interoperabilidad con otros contratos inteligentes.

Asimismo, existe una multitud de proyectos y plataformas alternativas a Ethereum para desplegar contratos inteligentes, algunos con funcionalidades adicionales o enfoques alternativos. Muchas de estas plataformas son compatibles con la EVM, por lo que, en general, el código de los contratos puede ser portado de manera relativamente sencilla de una plataforma a otra (Leal, 2023).

Ethereum es una *blockchain* pública que permite la participación de cualquier persona. No obstante, también existe la posibilidad de desplegar una *blockchain* privada. Una *blockchain* privada es aquella que es mantenida por una organización específica, la cual tiene control sobre el proceso de minado y el algoritmo de consenso. (Anwar, 2021). En la siguiente sección se profundiza al respecto.

2.4 Estado del Arte

Desde la popularización de Bitcoin, Ethereum y otras plataformas similares, la tecnología *blockchain* ha experimentado importantes avances en varios aspectos. En este trabajo, se analizarán dos aspectos relevantes: los algoritmos de consenso y las *blockchains* privadas empresariales.

2.4.1 Algoritmos de consenso

Los algoritmos de consenso desempeñan un papel fundamental en la tecnología *blockchain*, al permitir a los participantes llegar a un acuerdo sobre el estado del sistema distribuido. A continuación, se presentan algunos de los algoritmos de consenso más conocidos y utilizados:

1. Proof of Work

En 1992, Dwork y Naor proponen un sistema para limitar el correo electrónico no deseado, comúnmente conocido como *spam*.

El sistema consiste en que el remitente del mensaje realice una determinada cantidad de cálculos computacionales y proporcione una prueba de haber completado dichos cálculos o trabajo computacional. Si un correo electrónico no incluye una prueba de haber realizado ese trabajo computacional, el mensaje es automáticamente descartado por el destinatario.

Este proceso es posible gracias a las funciones hash, descritas en la sección anterior. Obtener el resultado de una función hash a partir de su preimagen, o argumento, es una operación que no implica un costo computacional considerable. Sin embargo, encontrar la preimagen de una función hash a partir de su imagen, o resultado, es costoso, ya que no existen mecanismos para hacer ingeniería inversa a las reglas de funcionamiento interno de la función hash, por lo tanto, es necesario obtener el resultado por fuerza bruta (Katz & Lindell, 2007).

Más tarde, en 1997, Adam Back implementa HashCash, un sistema que utiliza una técnica distinta, pero que sigue el mismo concepto general propuesto por Dwork y Naor (HashCash, s.f.). En el correo enviado por Back (1997) para publicar su trabajo, se menciona la expresión “proof of work”.

Más tarde, Back (2002) publica un trabajo donde propone usos adicionales a su sistema HashCash, entre ellos, se propone usar Proof of Work como mecanismo de *minting* (acuñación) para b-money, un sistema de de dinero digital propuesto por Wei Dai (1998).

Luego, el mecanismo Proof of Work es elegido por Satoshi Nakamoto (2008) como mecanismo de consenso para Bitcoin.

La principal desventaja que de este mecanismo de consenso es su elevado costo energético (Ante & Fiedler, 2021). Los mineros deben realizar cálculos constantemente para lograr una colisión de hash y así poder minar un nuevo bloque, lo que les otorga una recompensa en bitcoins.

Se ha vuelto común la fabricación de hardware dedicado exclusivamente a la minería de Bitcoin y otras criptomonedas. En particular, existe un mercado considerable de equipos conocidos como ASICs[11], que son dispositivos electrónicos con circuitos diseñados específicamente para la minería de Bitcoin. Estos dispositivos logran niveles de eficiencia mucho mayores que los computadores de propósito general (Pathirana & Halgamuge, 2019).

Sin embargo, esta tendencia puede conducir a una centralización de la actividad minera en manos de unos pocos actores o empresas especializadas, lo cual podría representar un riesgo para la seguridad de las criptomonedas (Kleine, 2019).

En respuesta a esta situación, algunas criptomonedas han realizado modificaciones en sus mecanismos de consenso para hacerlos resistentes a los ASIC (Cant, 2019). Estos mecanismos implementan cambios que hacen que la minería de criptomonedas utilizando ASICs sea imposible (getmonero.org, s.f.).

2. Proof of Stake

Dadas las desventajas mencionadas anteriormente del mecanismo de consenso Proof of Work, ha surgido el mecanismo Proof of Stake (prueba de participación) como una alternativa prometedora.

Este mecanismo de consenso consiste en que los nodos que desean validar transacciones y crear nuevos bloques deben demostrar la posesión de una cierta cantidad de la criptomoneda nativa de la red. Esta cantidad de criptomoneda se denomina "participación" o "apuesta". Cuanto mayor sea la participación de un nodo, más probabilidades tendrá de ser seleccionado para forjar un nuevo bloque y recibir recompensas. (Kashyap, 2023)

[11] ASIC: *Application-Specific Integrated Circuit*

El proceso de selección se realiza de manera pseudoaleatoria y ponderada, teniendo en cuenta la cantidad de participación que cada nodo tiene (Nguyen, et al., 2019).

Al utilizar el mecanismo Proof of Stake, se espera lograr varias ventajas sobre el Proof of Work. Algunas de estas incluyen:

1. Eficiencia energética: El Proof of Stake no requiere la realización de cálculos intensivos, lo que reduce significativamente el consumo de energía en comparación con el Proof of Work (Kashyap, 2023).

2. Mayor escalabilidad: Al eliminar la necesidad de competir por la resolución de problemas, el mecanismo *Proof of Stake* permite un mayor número de transacciones por segundo y un procesamiento más rápido de las transacciones en la red (de Isidro, Anderson, & Reddy, 2022).

3. Mayor seguridad: Los ataques de doble gasto y ataques del 51% son más difíciles de llevar a cabo en un sistema de *Proof of Stake*, ya que un atacante necesitaría controlar la mayoría de la participación en la red, lo que resulta costoso y poco probable (Kashyap, 2023).

4. Incentivos económicos alineados: Al requerir que los nodos posean y apuesten su criptomoneda nativa, el mecanismo *Proof of Stake* alinea los intereses económicos de los participantes con el buen funcionamiento y la seguridad de la red. Si un nodo actúa de manera maliciosa, puede perder parte o la totalidad de su participación, lo que desincentiva el comportamiento perjudicial (Kashyap, 2023).

Vitalik Buterin, el creador de Ethereum respalda la idea de que el mecanismo Proof of Stake es superior al Proof of Work, argumentando que el Proof of Stake "proporciona más seguridad al mismo costo" (Buterin, 2020).

Basándose en estas consideraciones, Ethereum ha tomado la decisión de cambiar su mecanismo de consenso a Proof of Stake. Esta transición representa un hito significativo en el desarrollo de Ethereum. Se espera que este cambio brinde beneficios en términos de eficiencia energética, escalabilidad y participación comunitaria en el proceso de validación de transacciones y forja de bloques.

3. Proof of Authority

En el ámbito de las *blockchains* privadas, donde la descentralización completa no es necesaria, se suelen utilizar mecanismos de consenso como la Prueba de Autoridad (*Proof of Authority*) (Hay, 2021).

La Prueba de Autoridad es un algoritmo de consenso en el que la capacidad de generar nuevos bloques y validar transacciones está basada en la identidad y reputación de los

nodos autorizados en lugar de en el poder de cálculo o la participación económica, como ocurre en *Proof of Work* y *Proof of Stake*.

En una red *Proof of Authority*, se designa a un grupo limitado de nodos conocidos como "autoridades" o "validadores" que tienen la responsabilidad de crear y validar nuevos bloques. Estas autoridades son seleccionadas y reconocidas por su reputación, confiabilidad y autoridad en el sistema. Por lo general, estas *blockchains* privadas son gestionadas por una organización o un consorcio de entidades que confían mutuamente entre sí.

La Prueba de Autoridad ofrece beneficios en términos de eficiencia y escalabilidad, ya que no requiere una gran cantidad de poder de cómputo ni recursos energéticos como en el caso de *Proof of Work*. Además, la red *Proof of Authority* es más rápida en términos de confirmación de transacciones, ya que no hay necesidad de esperar a que se realicen complicados cálculos matemáticos o que los nodos compitan por resolver problemas criptográficos (Baliga, Subhod, Kamat, & Chatterjee, 2018; Hay, 2021).

Dentro del marco de consenso basado en Proof of Stake, existen diversas variantes destacadas, entre las cuales se encuentran Raft e IBFT. La distinción principal entre estas radica en su modelo de fallas: Mientras que IBFT es tolerante a nodos que operan de manera maliciosa, Raft está diseñado solo para tolerar fallos accidentales.

El problema de garantizar la tolerancia del sistema ante participantes maliciosos es comúnmente conocido como el problema de los generales bizantinos (Lamport, Shostak, & Pease, 2016). Se dice que los sistemas que pueden resistir y mitigar los efectos de nodos maliciosos son “tolerantes a fallas bizantinas” (Driscoll, Hall, Sivencrona, & Zumsteg, 2003).

A continuación, se listan las diferencias más importantes entre Raft e IBFT.

Modelos de fallas:

- **Raft**: El modelo de falla de Raft asume un escenario de falla benigna, donde los nodos pueden fallar, pero no se comportan maliciosamente. Se basa en la elección de líderes para coordinar las operaciones y utiliza un enfoque de elección periódica para asegurar que se elija un líder válido (Baliga, Subhod, Kamat, & Chatterjee, 2018).

- **IBFT**: El modelo de falla de IBFT es más general y puede manejar fallas bizantinas, lo que significa que los nodos pueden fallar o incluso actuar de manera maliciosa. Utiliza técnicas criptográficas y acuerdos en dos etapas para garantizar el consenso incluso en presencia de comportamientos adversos (Baliga, Subhod, Kamat, & Chatterjee, 2018).

Fases del algoritmo:

- **Raft**: Raft se compone de tres fases principales: elección de líder, replicación de registros y confirmación de registros. La elección de líder se realiza a través de un proceso de votación, mientras que la replicación y confirmación de registros se realizan mediante la comunicación entre los nodos (Baliga, Subhod, Kamat, & Chatterjee, 2018).
- **IBFT**: IBFT se compone de cuatro fases principales: Pre-prepare, prepare, y commit. Estas fases implican intercambios de mensajes entre los nodos para proponer, validar y confirmar transacciones (Baliga, Subhod, Kamat, & Chatterjee, 2018).

Rendimiento:

Según estudios realizados por Baliga, Subhod, Kamat y Chatterjee (2018), se ha observado que el mecanismo de consenso Raft demuestra un mejor rendimiento en condiciones de cargas de trabajo elevadas, superando las 1650 transacciones por segundo. Por otro lado, el mecanismo de consenso IBFT muestra un rendimiento superior en cargas de trabajo más bajas.

2.4.2 *Blockchains* Privadas empresariales

Las *blockchains* privadas empresariales son utilizadas por organizaciones y consorcios con el objetivo de mejorar la eficiencia, transparencia y seguridad de sus operaciones internas. Estas *blockchains* se implementan en un entorno controlado y restringido, donde los participantes son conocidos y confían entre sí. En este contexto, los mecanismos de consenso como *Proof of Authority* pueden ser adecuados debido a sus características particulares (Consensys, 2018).

En una *blockchain* privada empresarial, las autoridades o validadores son seleccionados y designados por la organización o el consorcio. Estas entidades son confiables y tienen un historial probado, lo que minimiza el riesgo de comportamiento malicioso o acciones perjudiciales para la red. Al depositar la confianza en estas autoridades, se simplifica el proceso de consenso y se logra una mayor eficiencia en la validación de transacciones (Baliga, Subhod, Kamat, & Chatterjee, 2018).

Además, en el ámbito empresarial, la prioridad suele ser la eficiencia y la escalabilidad, ya que se busca agilizar las operaciones internas y minimizar los costos. Los mecanismos de consenso como *Proof of Work* o *Proof of Stake*, que requieren un alto consumo de energía y recursos computacionales, pueden resultar innecesarios y poco prácticos en este contexto. Por lo tanto, *Proof of Authority* se convierte en una opción más adecuada al ofrecer una mayor eficiencia y velocidad de confirmación de transacciones (Consensys, 2018).

A continuación, se expone un ejemplo del uso de una *blockchain* privada en el mundo empresarial.

Caso Walmart

En la industria alimentaria, la trazabilidad de los alimentos es un aspecto fundamental.

Es necesario rastrear el origen, el procesamiento y la distribución de los productos alimentarios a lo largo de toda la cadena de suministro. Esto implica identificar y documentar cada etapa del proceso, desde la producción primaria hasta llegar al consumidor final (Sristy, 2021).

La trazabilidad permite identificar rápidamente la fuente de cualquier problema de seguridad alimentaria, como brotes de enfermedades transmitidas por alimentos o contaminaciones. Esto facilita la retirada eficiente de los productos afectados del mercado y ayuda a prevenir riesgos para la salud pública (Hyperledger, 2019).

Sin embargo, implementar sistemas de trazabilidad en esta industria es una tarea compleja, incluso para empresas de la talla de Walmart. Por esta razón, Walmart e IBM vieron en la tecnología *blockchain* una oportunidad para agilizar los sistemas de trazabilidad (Hyperledger, 2019).

En primer lugar, llevaron a cabo pruebas de concepto para dos cadenas de suministro con el objetivo de probar el sistema. Un proyecto se centró en mejorar la trazabilidad de los mangos vendidos por Walmart en Estados Unidos, mientras que el otro buscaba rastrear la carne de cerdo vendida en China (Hyperledger, 2019). El sistema de trazabilidad existente era extremadamente lento, ya que se requerían alrededor de siete días para revisar el origen de un producto específico (Hyperledger, 2019).

Las pruebas de concepto, basadas en Hyperledger Quorum, resultaron exitosas y lograron agilizar la trazabilidad de los productos, reduciendo el tiempo necesario para revisar el origen de un producto de siete días a tan solo 2.2 segundos (Hyperledger, 2019). A partir de estas pruebas exitosas, Walmart expandió su sistema de trazabilidad basado en *blockchain* a al menos 25 productos (Hyperledger, 2019).

2.5 Revisión sistemática

Se realizó una revisión sistemática con búsquedas en Web of Science, una plataforma de pago que provee acceso a una base de datos de publicaciones académicas.

2.5.1 Alcance

La revisión sistemática evalúa las investigaciones académicas existentes en el campo de las aplicaciones descentralizadas, específicamente el uso de *blockchain* como estructura de datos para el almacenamiento y distribución de información en redes descentralizadas.

2.5.2 Preguntas de revisión

Pregunta Principal

¿Es la tecnología *blockchain* usada para el almacenamiento y sincronización de datos en otras industrias (excluyendo criptomonedas)?

Preguntas secundarias

1. ¿Es la tecnología *blockchain* usada para el almacenamiento y sincronización de datos por proveedores de servicio de Internet?
2. ¿Cuáles industrias utilizan aplicaciones decentralizadas privadas y con propósitos específicos?
3. ¿Es la tecnología *blockchain* eficiente para el almacenamiento y la distribución de datos?

2.5.3 Cadenas de búsqueda:

Luego de una búsqueda exploratoria en Web of Science, se definieron 2 cadenas de búsqueda para la revisión sistemática:

Cadena 1:

((("Communications Service Provider" OR "Internet Provider" OR "Internet Service Provider" OR "ISP") AND (("Distributed networking" OR "decentralized application" OR "decentralized network" OR "distributed application" OR "distributed network") OR "blockchain")) OR (("configuration") AND ("Internet of Things" OR "IoT") AND (("Distributed networking" OR "decentralized application" OR "decentralized network" OR "distributed application" OR "distributed network") OR "blockchain")))

Cadena 2:

((("Communications Service Provider" OR "Internet Provider" OR "Internet Service Provider" OR "ISP") AND (("Distributed networking" OR "decentralized application" OR "decentralized network" OR "distributed application" OR "distributed network") OR "blockchain")) OR ((("Distributed networking" OR "decentralized application" OR

"decentralized network" OR "distributed application" OR "distributed network") OR "blockchain") AND (("configuration") AND ("edge computing"))))

2.5.4 Resultados

Para la cadena 1 se buscaron las palabras clave en el *abstract* de los artículos. Además, se seleccionó solo los artículos publicados entre 2018 y 2022. Esto arrojó 35 resultados.

Utilizando esta misma metodología, la cadena 2 entregó 9 resultados. Para incrementar el número de resultados, se buscó en todos los campos, lo cual arrojó 18 resultados.

Considerando el número y contenido de los artículos, se decide utilizar la cadena 1.

2.5.5 Selección

Interesa saber cuál es el uso que se le da a los datos: si el funcionamiento de los dispositivos *endpoint* (nodos de la red que están desplegados en terreno) depende de la información obtenida desde la *blockchain*, o si los dispositivos simplemente añaden información a la *blockchain* para que esta sea analizada posteriormente por entes externos.

Criterios de inclusión:

1. “Describe el comportamiento los *endpoints* como dependiente de los datos de la *blockchain*”.

2. “Describe los nodos de la red como configurables, programables, o menciona que estos tienen varias funciones”.

Criterios de exclusión:

1. Artículos no publicados en los últimos cinco años.

2. Artículos no accesibles para el equipo de investigación.

3. Artículos no escritos en inglés o en español.

2.1.6 Resultados

De los 35 artículos, 6 fueron seleccionados según los criterios de inclusión. 4 fueron rechazados por los criterios de exclusión mencionados. Cabe destacar que los artículos no publicados en los últimos cinco años fueron descartados durante la búsqueda en Web of Science, por lo tanto, no están incluidos en los 35 resultados.

Los artículos seleccionados fueron:

1. Management and Monitoring of IoT Devices Using Blockchain (Košťál, Helebrandt, Belluš, Ries, & Kotuliak, 2019)
2. Integrating blockchain and Internet of Things systems: A systematic review on objectives and designs (Nguyen, Babar, & Boan, 2021)
3. BorderChain: Blockchain-Based Access Control Framework for the Internet of Things Endpoint (Oktian, 2021)
4. Blockchain Expansion to secure Assets with Fog Node on special Duty (Gul, Rehman, & Paul, 2020)
5. Application-Aware Consensus Management for Software-Defined Intelligent Blockchain in IoT (Wu, Dong, Ota, Li, & Yang, 2020)
6. A blockchain security scheme to support fog-based internet of things (Mohapatra, Bhoi, Jena, Nayak, & Singh, 2022)

2.5.7 Hallazgos destacables

1. Las *blockchain*s privadas son viables en la práctica (Košťál et al, 2019).

2. Se puede autenticar a los administradores legítimos de la *blockchain*, y autorizarlos a agregar cambios a las configuraciones de los dispositivos de la red mediante certificados digitales. Todo esto sin necesitar infraestructura de llave pública (PKI[12]) (Košťál et al, 2019).

3. Se puede trabajar de forma segura sin necesidad de una base de datos *off-chain*, es decir, una base de datos externa a la cual se la haga referencia desde la *blockchain*. Sin embargo, si por alguna razón se desea, se puede utilizar este tipo de base de datos. (Košťál et al, 2019).

4. *Blockchain* es útil para añadir trazabilidad a la configuración de computadores luego de una intrusión informática no autorizada, permitiendo detectar cambios ilegítimos a estos archivos. (Yustus Eko Oktian, Sang-Gon Lee, 2021).

2.5.8 Síntesis de la revisión sistemática

Utilizar la primera cadena de búsqueda arrojó una mayor cantidad de resultados, ya que la intersección de IoT y *blockchain* es un campo de gran interés en la actualidad, pero también hizo que prácticamente todos los resultados sean referentes a IoT.

[12] PKI: *Public Key Infrastructure*

Esto último obliga a excluir todos aquellos artículos cuyos hallazgos no sean aplicables al proyecto. Para esto, se utilizaron criterios de selección relativos a los problemas abordados en los artículos.

Además, el criterio de extracción de datos referente a la dirección del flujo de información es vital para poder evaluar la aplicabilidad de los hallazgos en el proyecto.

En general, los artículos arrojados por la búsqueda se centran en almacenar información en la *blockchain*. Košťál et al (2019) abordan un problema bastante similar al de este proyecto.

La Tabla 1 resume los hallazgos respecto a las preguntas de investigación

Tabla 1: Hallazgos respecto a las preguntas de investigación

Pregunta	**Hallazgos**
¿Es la tecnología *blockchain* usada para el almacenamiento y sincronización de datos en otras industrias (excluyendo criptomonedas)?	Según los análisis de acercamiento realizados con los resultados de las dos cadenas de búsqueda, así como con las búsquedas no estructuradas, se concluye que existe abundancia de artículos que proponen el uso de *blockchain* en toda clase de industrias. Sin embargo, los artículos encontrados no hablan sobre los resultados de la implementación práctica de estas investigaciones en la industria. En general, los artículos son solamente propuestas y documentos de diseño. La revisión sistemática incluida en la lista de artículos seleccionados también habla sobre el diseño y los objetivos, no evalúa resultados (Nguyen, Babar, & Boan, 2021).
¿Cuáles industrias utilizan aplicaciones decentralizadas privadas y con propósitos específicos?	Las investigaciones propositivas mencionadas anteriormente aplican para muchísimas industrias, incluyendo: Internet of Things, Internet of Vehicles, seguridad en la nube, redes definidas por software (SDN). Sin embargo, no se encontró evidencia del uso de *blockchain* en la práctica.

¿Es la tecnología *blockchain* usada para el almacenamiento y sincronización de datos por proveedores de servicio de Internet?	Teniendo en cuenta las mismas limitaciones de las preguntas anteriores, se encontró artículos que proponen el uso de *blockchain* en contextos de CSP o ISP, sin embargo, no fueron seleccionados. Los artículos revisados de manera preliminar que caían dentro de este contexto se referían principalmente a SDN (Software Defined Networking), redes definidas por software.
¿Es la tecnología *blockchain* eficiente para el almacenamiento y la distribución de datos?	Sí. Las propuestas analizadas, así como la revisión sistemática dentro de los artículos seleccionados, demuestran que la *blockchain* es una forma eficiente y teóricamente viable de almacenar y distribuir datos.

3. PLANIFICACIÓN DEL PROYECTO

3.1 Tecnologías a utilizar

3.1.1 Exploración del entorno

Se realizó una exploración del entorno de librerías, lenguajes, *frameworks* y herramientas utilizadas en la industria.

Durante este proceso, se determinó que el factor decisivo para seleccionar las tecnologías a utilizar sería la capacidad de realizar transacciones privadas. Dado que Ethereum, así como la mayoría de las *blockchains* disponibles, carece de funciones de privacidad incorporadas de forma nativa, se elaboró una lista de opciones que sí ofrecían esta capacidad. A continuación, se realizó una comparación entre ellas y se seleccionó una plataforma de *blockchain* en función de esta evaluación.

La Tabla 7 muestra las herramientas y las observaciones

Tabla 2: comparativa de plataformas *blockchain* con privacidad

Plataforma	**Observaciones**
Hyperledger Fabric: plataforma *blockchain* de código abierto desarrollada por la Linux Foundation. (HyperLedger, 2022)	Proporciona un entorno configurable que permite crear una red *blockchain* privada. Además, cuenta con características de privacidad y control de acceso que permiten mantener la información confidencial solo entre las partes autorizadas. Sin embargo, la forma que tiene para lograr confidencialidad entre las partes autorizadas es solamente mediante “grupos”. Cualquier nodo que pertenezca al grupo puede ver la información transmitida. Esto significa que habría que crear un nuevo grupo para cada Nodo, lo cual sería más ineficiente, e implicaría desarrollar un sistema que cree los grupos de forma dinámica cada vez que se quiera agregar un nuevo dispositivo a la red.

ConsenSys Quorum: Una implementación de Ethereum que proporciona características de privacidad, diseñada específicamente para redes privadas y consorcios empresariales. (Consensys Quorum, 2023)	Las dos funcionalidades principales que ofrece en cuanto a privacidad son las *permissioned networks* (redes con permisos), que solo permiten a ciertos nodos autorizados unirse a la red, efectivamente logrando que toda la *blockchain* sea privada, y las transacciones privadas, donde la información de una transacción específica es privada, sin afectar la privacidad de las demás transacciones de la *blockchain.*
Corda: Es una plataforma blockchain desarrollada por la empresa R3. Corda está diseñada para aplicaciones empresariales y permite la creación de redes blockchain privadas. autenticidad de las transacciones. (Corda, s.f.)	Proporciona un modelo de datos compartidos que permite la confidencialidad de la información. Posee una versión gratuita y una versión de pago.

4.1.2 Plataforma elegida

Se decide utilizar Quorum debido a su naturaleza de código abierto. Esta elección permitirá a Grupo GTD extender el proyecto con funcionalidades adicionales en el futuro sin depender de un proveedor de pago, a diferencia de Corda.

Además, Quorum ofrece privacidad a nivel de transacciones, en contraste con HyperLedger Fabric, cuyas funcionalidades de privacidad se limitan a nivel de grupos. La posibilidad de contar con este nivel de granularidad en la privacidad hará que el sistema sea más manejable y que el desarrollo de software sea más sencillo.

Con el fin de asegurar la seguridad y privacidad de las transacciones en la red, se emplea la funcionalidad de transacciones privadas de Quorum. Esto resulta especialmente útil para proteger información sensible que pueda ser transmitida a través de los archivos de configuración.

4.1.3 Herramientas adicionales

Web3.js

Web3.js es una librería de Javascript que permite interactuar con nodos de *blockchain*, desplegar contratos inteligentes, enviar transacciones, leer y escribir datos en la *blockchain*, y realizar otras operaciones relacionadas con la *blockchain*.

Git

Sistema de control de versiones utilizado para versionar el código de la aplicación.

Solidity

Lenguaje de programación utilizado para programar el contrato inteligente.

Truffle

Truffle es un *framework* popular en el desarrollo de aplicaciones descentralizadas (DApps) en la plataforma Ethereum. Proporciona un conjunto de herramientas y bibliotecas que facilitan la creación, prueba e implementación de contratos inteligentes en Ethereum. Inicialmente se utilizó Truffle para compilar y desplegar los contratos, e interactuar con ellos, pero debido a lo desactualizado que está su soporte para Quorum, lo cual desencadenó en varios problemas técnicos, se decide prescindir de Truffle, y continuar el desarrollo del proyecto utilizando solamente Web3.js para la compilación y despliegue del contrato, así como la posterior interacción con este.

4. TOMA DE REQUISITOS

Para la toma de requisitos, se utilizó la metodología *Venture Design*, una técnica de prototipado ágil. De esta forma, se proponen hipótesis sobre los requisitos de los usuarios y luego, a través de entrevistas con personas reales, se confirman o refutan aquellas hipótesis.

4.1 Declaración de posicionamiento del producto

Usando la técnica propuesta por Geoffrey More (1991), se plantea de forma resumida una declaración que especifica qué producto se quiere desarrollar, sus usuarios y diferencias con las soluciones existentes. Se le da a la propuesta el nombre de ConfigChain (combinación de *configuration* y *blockchain)*:

"Para las empresas proveedoras de servicios de comunicaciones, quienes necesitan gestionar las configuraciones de los equipos de red en las dependencias de sus clientes, *ConfigChain* es un sistema informático que permite distribuir las configuraciones de los equipos de red que entregan servicio a los clientes.

A diferencia de las soluciones centralizadas con arquitectura cliente-servidor, ConfigChain utiliza una red distribuida utilizando *blockchain*, lo que otorga resiliencia y trazabilidad al sistema".

4.2 Hipótesis de personajes

Se plantearon dos personajes ficticios que representan a los usuarios del sistema actual de configuración de VSB. Estos se detallan en la Tabla 2.

Tabla 2: Personajes

Nombre	Rol
Manuel	Ingeniero del área de operaciones de Grupo GTD
Luciano	Ingeniero del área de desarrollo de Grupo GTD

Para cada personaje, se crea una hipótesis sobre su percepción del sistema de configuración de VSB existente, utilizando el modelo conocido como *Think, See, Feel, Do*. En este modelo, se lista para cada personaje lo que este piensa, ve, siente, y hace.

Manuel, Ingeniero del área de operaciones de Grupo GTD:

Lo que piensa: El sistema de gestión de configuración de equipos funciona, pero es poco confiable, muy susceptible a ataques informáticos y no guarda registro de los cambios.

Lo que ve: Problemas comunes con la gestión de configuración. Por ejemplo: que se le entregue a un cliente un equipo que no ha sido configurado, y que no se pueda entregar correctamente el servicio, porque el equipo tiene la configuración del cliente anterior que lo utilizó.

Lo que siente: Que estos problemas le quitan tiempo valioso que podría destinarse a otras cosas. Que el sistema actual es insuficiente.

Lo que hace: Asegurarse de que el servicio se entregue correctamente a todos los clientes.

Luciano, Ingeniero de desarrollo de Grupo GTD:

Lo que piensa: Que el sistema de configuración VSB es poco confiable.

Lo que ve: Que, al desarrollar software para el sistema de gestión de configuraciones, debe realizar scripts sin mucha estructura.

Lo que siente: Que no se ha puesto suficiente atención en la gestión de configuraciones de los VSB.

Lo que hace: Desarrollar software de todo tipo para satisfacer las necesidades de la empresa.

4.3 Propuesta de valor

En la Tabla 4, se compara el estado actual con la propuesta del proyecto, demostrando el valor que se añade mediante el desarrollo de este.

Tabla 4: Propuesta de valor

Escenarios problemas	**Alternativas actuales**	**Propuesta de valor**
Un cliente empresa contacta a la compañía porque una de sus subredes no está entregando servicio de Telefonía sobre IP (VoIP).	Que un ingeniero de operaciones ingrese al equipo VSB manualmente mediante SSH para revisar que la configuración sea la correcta. Que un técnico acuda a las dependencias del cliente con una nueva máquina pre-configurada, retire la antigua e instale la máquina nueva.	Realizar la revisión de configuración y actualización de configuración usando un flujo definido para ello, sin necesidad de conectarse al equipo. Revisar el contenido de la *blockchain* para ver cuál ha sido la última configuración de ese VSB.
Ocurre un ataque cibernético y no se sabe qué equipos aún están afectados.	Se escribe un script *ad hoc* que se conecta por SSH para ir equipo por equipo borrando los archivos maliciosos y reemplazando.	Revisar el contenido de la *blockchain* para ver si ha habido modificaciones maliciosas. Agregar datos a la *blockchain* para actualizar archivos.
Se necesita instalar un VSB en las dependencias de un cliente.	Se ejecuta un script que carga la configuración en el VSB, antes o después de haberlo instalado en las dependencias del cliente.	Se agrega un bloque a la *blockchain*, que contiene toda la configuración relevante. El VSB, al ser activado, descarga la *blockchain* y detecta que existe una configuración disponible para él.

4.4 Entrevistas a personas reales

Para obtener una evaluación precisa de la percepción de los trabajadores de Grupo GTD sobre el sistema actual que gestiona los equipos VSB, se llevaron a cabo entrevistas a dos ingenieros de la empresa que han tenido interacciones con dicho sistema.

La Tabla 5 detalla la entrevista al ingeniero de desarrollo de Grupo GTD.

Tabla 5: Entrevista al ingeniero de desarrollo

N°	**Pregunta**	**Respuesta**
1	De forma muy resumida: ¿Cuál es su función dentro de la compañía?	Ingeniero de Desarrollo de Software
2	¿Cada cuánto realiza tareas relacionadas con los VSB? ¿Qué tareas?	De forma esporádica. Sólo cuando existen solicitudes de nuevas funcionalidades o reporte de errores críticos.
3	Cuando su equipo debe escribir software para los VSB o sus sistemas adyacentes ¿Cree que estos sistemas y procesos están diseñados de forma adecuada?	La construcción de nuevas funcionalidades se apega a las prácticas y procesos del área de desarrollo, por ende, son diseños ya revisados y ajustados a las necesidades del Grupo GTD.
4	¿Qué piensa sobre el sistema actual de gestión de configuraciones de los VSB?	Actualmente, la gestión de configuraciones cumple con el objetivo principal que es aprovisionar de forma semi-automática a los VSB. Las dificultades están expresadas en la falta de trazabilidad y recuperación de configuraciones en caso de eventos catastróficos.
5	¿Cree que la forma de actualizar la configuración de los VSB es la adecuada?	No. La actualización de configuraciones podría ser mucho más óptima.
6	¿Cuáles son los problemas típicos que tiene la compañía respecto al	El problema recurrente es frente a la falla de un VSB. Si se requiere reemplazar el

	actual sistema de gestión de configuraciones de VSB?	dispositivo, se deben volver a aplicar todas las configuraciones de forma manual.
7	¿Cree que la trazabilidad de las configuraciones es importante?	Sí, ya que permitiría la automatización de varios procesos internos.
8	¿Algún comentario adicional sobre los VSB y sus sistemas adjuntos?	El producto VSB debe ser mirado como un caso de uso respecto a los sistemas de aprovisionamiento y orquestación del Grupo GTD. La experiencia que se logre adquirir a través de este trabajo de Investigación, serán potencialmente utilizados en otros proyectos. Por ejemplo, plataformas de IoT, SmartCity, etc.

Por otro lado, la Tabla 6 muestra las respuestas del Ingeniero de operaciones.

Tabla 6: Entrevista al ingeniero de operaciones

N°	**Pregunta**	**Respuesta**
1	De forma muy resumida: ¿Cuál es su función dentro de la compañía?	Trabajo en el área de operaciones manteniendo el equipamiento de la red.
2	¿Cada cuánto realiza tareas relacionadas con los VSB? ¿Qué tareas?	Semanalmente, mantengo y actualizo los equipos VSB de los clientes.
3	Cuando su equipo debe escribir software para los VSB o sus sistemas adyacentes ¿Cree que estos sistemas y procesos están diseñados de forma adecuada?	2 horas

4	¿Qué piensa sobre el sistema actual de gestión de configuraciones de los VSB?	El sistema actual es poco intuitivo, poco amigable con el usuario y es difícil gestionar el equipo a través de ese sistema.
5	¿Cree que la forma de actualizar la configuración de los VSB es la adecuada?	No, esperaría que fuese más rápida.
6	¿Cuáles son los problemas típicos que tiene la compañía respecto al actual sistema de gestión de configuraciones de VSB?	Sistema lento, poca capacidad de troubleshooting online.
7	¿Cree que la trazabilidad de las configuraciones es importante?	Sí, nos permitiría tener la información de las últimas modificaciones hechas antes de un incidente con el equipo.
8	¿Algún comentario adicional sobre los VSB y sus sistemas adjuntos?	Espero que se pueda mejorar el sistema de gestión para dar un servicio más ágil a nuestros clientes.

4.5 Evaluación de hipótesis

Se confirman las hipótesis. Los entrevistados coinciden en que el sistema actual es poco conveniente y las formas que proponen para mejorarlo se alinean con las propuestas del proyecto.

Además, se nota un interés de las personas entrevistadas por la mejoría en la experiencia de usuario en general, especialmente por la facilidad de uso y automatización, y no solo por las características específicas que son foco del proyecto, como la trazabilidad.

4.6 Historias de usuario

Al confirmar las hipótesis sobre las necesidades de los usuarios, se decide definir las historias de usuario que se muestran en la Tabla 7.

Tabla 7: Requisitos de usuario

ID	**Enunciado de la historia**	**Criterios de Aceptación**
H01	Como ingeniero, necesito un sistema que se ejecute en un computador de escritorio, con la finalidad de poder interactuar cómodamente con el sistema desde mi lugar de trabajo.	Dado que el ingeniero necesita poder interactuar cómodamente con el sistema desde su lugar de trabajo, entonces se desarrollará un sistema que se pueda ejecutar en sistemas operativos Windows, Linux o ambos.
H02	Como ingeniero de red, quiero poder cargar un archivo de configuración en el sistema para enviarlo a un nodo de la *blockchain*.	Dado que el ingeniero necesita poder cargar un archivo de configuración en el sistema para enviarlo a un nodo, el sistema debe tener la capacidad de leer archivos locales e incluir sus contenidos en la transacción
H03	Como ingeniero de red, quiero poder ejecutar manualmente una sincronización y descarga de la configuración al estar conectado a un nodo, con la finalidad de forzar al nodo a que descargue el archivo si ocurre algún error en la descarga automática.	Dado que el ingeniero necesita poder forzar al nodo a que descargue el archivo si ocurre algún error en la descarga automática, el sistema debe permitir obtener los datos de la *blockchain* manualmente.
H04	Como ingeniero de red, quiero recibir una confirmación de que el archivo de configuración ha sido enviado correctamente a la red, para estar seguro de que mis cambios locales se propagan en la red.	Dado que el ingeniero necesita estar seguro de que sus cambios locales se propagan en la red, el sistema debe informar cuando un cambio de configuración se realiza exitosamente.
H04	Como ingeniero de red, quiero poder actualizar un archivo de	Dado que el ingeniero necesita arreglar errores en la entrega del servicio, el

	configuración existente en un nodo de la *blockchain*, para arreglar errores en la entrega del servicio.	sistema debe poder actualizar y sobreescribir un archivo de configuración existente.
H06	Como ingeniero, quiero poder ver el historial de versiones de los archivos de configuración en un nodo específico, con la finalidad de poder diagnosticar problemas y recuperar el sistema cuando ocurran fallos.	Dado que el ingeniero requiere poder diagnosticar problemas y recuperar el sistema cuando ocurran fallos, el sistema debe permitirle revisar los archivos de configuración anteriores.

4.7 Consideraciones no funcionales

4.1 Seguridad

4.1.1 Confidencialidad

Parte de la información almacenada en los archivos de configuración es sensible y debe ser considerada como confidencial. Para esto, se debe implementar:

- Un esquema de cifrado de la información contenida en la *blockchain*, o bien
- Un esquema de acceso seguro a información contenida en una base de datos separada de la *blockchain*, que contenga esta información confidencial.

4.1.2 Autenticación

Es necesario que los VSB acepten únicamente los bloques emitidos por los administradores legítimos de la red y no por terceros que puedan interferir maliciosamente en esta. Esto hace necesaria la existencia de un sistema de autenticación que valide que la información sea emitida por un nodo autorizado.

En este sentido, existen tres opciones:

- El mecanismo de consenso solamente considerará cadenas de bloques como válidas cuando todos los bloques estén firmados digitalmente por el administrador legítimo.

- El contrato inteligente solo permitirá la realización de ciertas acciones de privilegios elevados a quien desplegó el contrato. Es decir, el contrato valida que quien esté enviando los archivos de configuración sea el dueño legítimo de la red.

- El mecanismo de consenso no requerirá firmas digitales en los bloques, pero cada VSB verificará que los bloques estén firmados, y si no lo están, no instalará la configuración almacenada en ese bloque.

4.2 Automatización

Una vez el VSB tiene asignado un identificador, la detección y verificación de un bloque que contenga información destinada al VSB, este debe iniciar un proceso de instalación automática del archivo de configuración correspondiente.

5. PROPUESTA

5.1 Solución propuesta

La presente investigación plantea un sistema informático descentralizado que distribuye las configuraciones a los VSB (ver Figura 5), de forma que el historial de configuraciones sea trazable, inmutable y auditable, además de hacer posible la automatización de la instalación de las configuraciones.

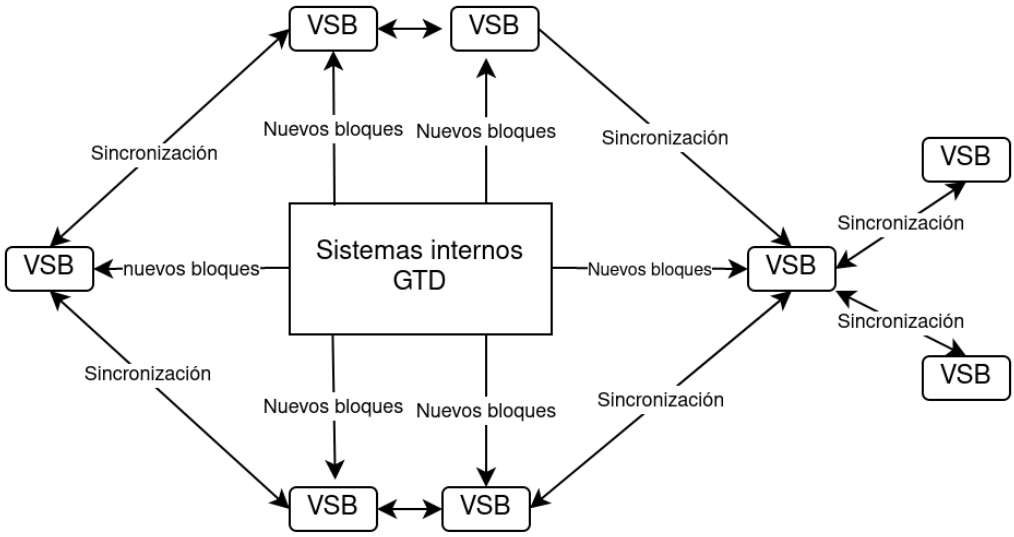

Figura 5: Sistema descentralizado de distribución de configuraciones

Gracias a la tecnología *blockchain*, no es necesario que los sistemas internos de Grupo GTD se comuniquen directamente con cada uno de los VSB al momento de enviar la configuración, sino que se emite una transacción en la *blockchain* que es difundida por la red en su conjunto, y al llegar al VSB correspondiente, este instala su configuración.

La propuesta, más específicamente, consiste en el despliegue de una red Quorum propia y exclusiva para Grupo GTD, cuya única finalidad es la sincronización de los archivos de configuración, cumpliendo los requisitos mencionados.

Sobre esta red, se despliega un contrato inteligente que implementa las funcionalidades necesarias para enviar y leer los datos, manteniendo la confidencialidad de la información transmitida.

Además, se detallan algunos procesos y consideraciones relacionadas a la preparación de los equipos VSB, necesarios para dejarlos en condiciones de poder unirse a la red de forma autónoma, recibir e instalar su configuración.

5.2 Tipos de nodos de la red

En la red propuesta existen tres tipos de nodos:

1. **Nodos autorizados:** Corresponden a computadores controlados en todo momento por Grupo GTD, y ubicados en las dependencias de la empresa.
2. **Nodos VSB**: Se ejecutan en cada equipo VSB de la red, que generalmente se encuentran ubicados en las dependencias de cada cliente. Si bien son controlados remotamente por Grupo GTD, se debe actuar bajo la suposición de que un agente adversario podría manipular el equipo e intentar obtener información confidencial de clientes distintos al correspondiente a ese equipo específico.
3. **Nodos validadores**: Nodos que forman parte de la red, son capaces de realizar transacciones públicas, pero su función es alcanzar el consenso de la red.

5.3 Contrato Inteligente

Se debe desarrollar un contrato que permita a uno o más nodos autorizados agregar, mediante transacciones, información a la *blockchain,* y a los nodos VSB leer la información publicada por los nodos autorizados.

Los nodos VSB solo deben poder descifrar el contenido de las transacciones que están destinados a cada uno de ellos. Esta confidencialidad se logra mediante el uso de transacciones privadas de Quorum.

Las transacciones privadas de Quorum no requieren ningún tipo de modificación en el código fuente del contrato, sino que los destinatarios privados deben ser especificados en el momento de realizar una llamada a una función en Quorum, una vez que el contrato ya está desplegado, con un atributo *PrivateFor* en los parámetros de la transacción.

En el contrato inteligente de la propuesta debe existir, al menos:

1. Una función para enviar una transacción con los archivos de configuración a la *blockchain.*
2. Una forma para que un VSB pueda, a partir de su número identificador (ID), obtener su archivo de configuración. Esto puede tomar una o varias llamadas a funciones.

5.4 Transmisión de la información

En el contrato inteligente se define una estructura de datos que almacena la información que va a ser difundida por los nodos autorizados. La información corresponde a los hashes de las transacciones donde se enviaron los archivos de configuración de los VSB, y a partir de estos hashes de transacción, el VSB correspondiente puede obtener su archivo de configuración.

5.5 Scripts de interacción con el contrato inteligente

Una vez que el contrato inteligente está desplegado, se necesita interactuar con él para poder ejecutar las funciones definidas. Esto se hace a través de la librería Web3.js.

Para conectarse a un nodo de Ethereum, Web3.js utiliza la interfaz RPC (*Remote Procedure Call*) proporcionada por el cliente de Ethereum.

El cliente Ethereum es el software que se ejecuta en el nodo y realiza las funciones definidas en el protocolo de Ethereum. A través de esta interfaz RPC, se pueden realizar llamadas a métodos específicos del contrato inteligente, como invocar funciones o enviar transacciones para modificar el estado del contrato.

La librería Web3.js también proporciona funcionalidades adicionales, como la gestión de claves y firmas digitales, la obtención de información sobre bloques y transacciones, y la interacción con contratos inteligentes personalizados.

De esta forma, la propuesta contempla la implementación de una serie de *scripts* que utilicen la biblioteca Web3.js para llamar las funciones del contrato inteligente descrito anteriormente, con el fin de leer y escribir los datos de configuración. El usuario final del sistema interactuará directamente con estos *scripts.*

5.6 Cliente de Ethereum

Como se mencionó, el cliente de Ethereum es la pieza de software encargada de ejecutar el protocolo Ethereum. Este cliente está constantemente validando transacciones, ejecutando los contratos inteligentes, manteniendo la cadena de bloques e interactuando con otros nodos.

El cliente de Ethereum más popular es go-ethereum, también conocido como Geth. Quorum incluye una versión modificada de Geth, capaz de interpretar los parámetros especiales con los que Quorum extiende el protocolo de Ethereum. Por lo tanto, este proyecto utiliza el cliente de Geth modificado, y desarrollado por Quorum.

Además, en Quorum se utiliza un mecanismo de consenso diferente al de Ethereum. El mecanismo de consenso de Quorum es Istanbul Byzantine Fault Tolerance (IBFT), y supone ventajas en las redes no públicas, como un menor tiempo para alcanzar el consenso, o el hecho de no necesitar cargar las cuentas de los nodos con Ether (ETH) para que estos puedan realizar transacciones.

5.7 Despliegue de red Quorum con transacciones privadas

En Ethereum, por defecto, todas las transacciones y su información asociada son públicas y transparentes. La solución que propone Quorum para otorgar privacidad es que cada participante que necesite recibir, o leer, información confidencial debe tener un par de claves criptográficas: una clave pública y una clave privada. Luego, el emisor cifra los datos utilizando la clave pública del destinatario, y el destinatario descifra los datos utilizando su clave privada.

Este esquema se conoce como criptografía de clave pública (Hellman, 1978), y es ampliamente utilizado en la computación moderna (CloudFlare, s.f.).

Sin embargo, Geth por sí solo no tiene la capacidad de descifrar transacciones, por lo que se hace necesario incluir un gestor de transacciones privadas (*private transaction manager*).

Cada nodo que requiera leer transacciones privadas en la *blockchain* debe estar asociado a un gestor de transacciones privadas para poder descifrar la información usando su clave privada.

En este proyecto se utiliza la herramienta Consensys Tessera, desarrollada por los mismos creadores de Quorum.

Para cada nodo, Tessera almacena su clave pública y privada, y cuando Geth detecta una transacción privada, delega a Tessera (también llamado “nodo” Tessera) la tarea de descifrar la transacción. Solo los nodos Tessera que tienen acceso a la clave privada correspondiente pueden descifrar y leer una transacción privada, esto incluye el emisor de la transacción, que siempre tiene acceso a su contenido.

Todo esto proporciona características de privacidad y confidencialidad en comparación con la red Ethereum estándar, donde toda la información de las transacciones es pública.

5.8 Procesos de negocio necesarios

Es necesario que Grupo GTD implemente una serie de procesos que permitan el funcionamiento de la propuesta, y así reemplazar el sistema existente.

5.8.1 Despliegue previo del contrato

La red y el contrato deben estar desplegados antes de la instalación del software del primer nodo. Esto porque se debe dejar especificada la dirección del contrato inteligente en un archivo del VSB. De esta forma, los dispositivos VSB pueden salir de la empresa con la dirección del contrato guardada en su sistema de archivos local.

5.8.2 Instalación de software en los VSB

Cada VSB debe ser preparado antes de salir a terreno y poder ser parte de la red. Esta preparación incluye:

1. Instalar el software necesario: Geth, Tessera, y sus respectivas dependencias.

2. Un archivo que almacene el número identificador (ID) del VSB, para que el VSB pueda posteriormente buscar su configuración utilizando ese ID.

3. Creación de una cuenta de Ethereum en el nodo Geth.

4. Creación del par de claves criptográficas en Tessera.

5.8.3 Gestión de claves públicas

Al momento de realizar una transacción privada, es necesario especificar la clave pública del nodo destinatario de esa transacción. Para esto, es necesario tener un registro local de las claves públicas de todos los nodos a los que se quiera enviar transacciones.

Es decir, el nodo autorizado debe tener una lista de las claves públicas correspondientes a cada VSB de la red.

6. IMPLEMENTACIÓN

6.1 Desplegar una red Quorum

Se despliega un entorno de pruebas: una red Quorum de cinco nodos que se ejecuta localmente en un computador, asignándole a cada nodo un puerto. Los nodos se comunican mediante la interfaz de *sockets*, tal como si estuvieran desplegados en una red de varios computadores.

El objetivo es replicar cómo sería el comportamiento de los nodos en una red distribuida, pero con la facilidad de manejo de una red local.

De los cinco nodos, tres tienen un gestor de transacciones privadas adjunto, es decir, un nodo Tessera. Los dos nodos restantes son nodos validadores, que contribuyen a alcanzar el consenso de la *blockchain*, son capaces de realizar transacciones, enviar y recibir información (aunque en esta propuesta no se contempla que los nodos validadores hagan transacciones), pero no pueden descifrar información de transacciones privadas.

De los tres nodos con un gestor de transacciones privadas, uno es el nodo autorizado, y los otros dos corresponden a nodos VSB.

El directorio raíz del entorno de pruebas contiene los siguientes directorios:

```
network/
|-- Node-0/
|-- Node-1/
|-- Node-2/
|-- Node-3/
|-- Node-4/
|-- Tessera-0/
|-- Tessera-1/
|-- Tessera-2/
|-- scripts/
|-- vendor/
|-- artifacts/
```

Primero, se usó la herramienta *quorum-genesis-tool*, como se puede ver en la Figura 6. Esta herramienta sirve para generar cuentas y archivos de configuración para todos los nodos Geth, y para generar el archivo de génesis, que especifica parámetros de la red, tales como las asignaciones de saldo inicial para los nodos, las reglas de consenso (en este caso, IBFT) y otros parámetros personalizados.

```
npx quorum-genesis-tool --consensus ibft --chainID 1337 --blockperiod 5
--requestTimeout 10 --epochLength 30000 --difficulty 1 --gasLimit '0xFFFFFF'
--coinbase '0x0000000000000000000000000000000000000000' --validators 5 --members
0 --bootnodes 0 --outputPath 'artifacts'
```

Figura 6: uso de herramienta *quorum-genesis-tool*

A continuación, se detalla el proceso de inicialización de los nodos en la red utilizando los archivos de configuración generados. En la Figura 7 se muestra cómo se ejecuta Geth en cada directorio.

```
geth --datadir data init data/genesis.json
```

Figura 7: inicialización de los nodos

Es importante destacar que esta inicialización no implica que los nodos se unan de inmediato a la red. En su lugar, a partir del archivo de génesis y los archivos de configuración, se generan los directorios y archivos necesarios para almacenar el contenido de la *blockchain*.

Además, en el directorio "scripts", se incluye un *script* que realiza esta inicialización automáticamente para todos los nodos del entorno de pruebas.

Los nodos cero, uno y dos tienen respectivamente los nodos Tessera cero, uno y dos como gestores de transacciones privadas adjuntos. El nodo cero es el nodo autorizado.

Para iniciar la red y empezar a agregar bloques, es necesario ejecutar primero los nodos Tessera (gestores de transacciones privadas) que estén asociados a algún nodo Geth. Para ello, se utiliza un script "start_node.sh" que se encuentra en el directorio de cada nodo Tessera.

A su vez, el directorio de cada nodo Geth contiene un *script* de inicio. Este *script* inicia Geth. De esta forma, Geth empieza a recibir y validar bloques, y a agregarlos a su copia local de la *blockchain*. Este script, además, indica a los nodos cero, uno y dos cuál es la ruta en el sistema de archivos donde se encuentra el nodo Tessera.

En la Figura 8 se lista el contenido del *script* "*start_node.sh"* para el nodo cero.

```
PRIVATE_CONFIG=/home/dasb/Code/cc/network/Tessera-0/tm.ipc geth --datadir data
--nodiscover --istanbul.blockperiod 5 --syncmode full --mine --miner.threads=1
--verbosity 5 --networkid 10 --http --http.addr 127.0.0.1 --http.port 22000
--http.api admin,db,eth,debug,miner,net,shh,txpool,personal,web3,quorum,istanbul
--emitcheckpoints --port 30300 --allow-insecure-unlock
```

Figura 8: script “start_node.sh” para el nodo cero

La variable de entorno *PRIVATE_CONFIG* indica a Geth cuál es la ruta donde se encuentra el gestor de transacciones privadas, es decir, el nodo Tessera correspondiente.

La opción *--http.port* indica el puerto que Geth expone para interactuar con él. A través de este puerto, los scripts luego pueden interactuar con el nodo usando Web3.js.

6.2 Desarrollar el contrato inteligente

El código fuente del contrato inteligente se detalla en la Figura 9.

El contrato define una variable llamada *owner*. En el constructor del contrato, se asigna a *owner* el valor de *msg.sender*, que corresponde a la cuenta de quién está desplegando el contrato. En otras palabras, *owner* representa al nodo autorizado.

Luego, se define un variable de tipo *mapping* con el nombre *idToTxHashes*. Un *mapping* es una estructura de datos equivalente a un diccionario en otros lenguajes de programación. En este caso, *idToTxHashes* asigna a cada número identificador (ID) de VSB un arreglo de hashes de transacción.

Dentro de ese arreglo, están los hashes de las transacciones que incluyen el contenido de los archivos de configuración en el orden en que fueron agregados. Es decir, cada uno de esos hashes corresponde a una transacción donde se ha enviado un archivo de configuración, y el último hash del arreglo apunta a la configuración considerada activa para un determinado VSB.

Luego, se define la función *sendConfig*. Esta función recibe como entrada los datos de configuración del VSB específico.

Como se observa, en ninguna parte de la función o del contrato se especifica que las funciones sean ejecutadas de forma pública o privada. La privacidad de las llamadas a funciones se debe especificar al momento de llamar la función, no al definirlas.

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.5.13;

contract configSender{

    //node authorized to publish blocks
    address private owner;

    constructor() public{
        owner = msg.sender;
    }

    // Takes the VSB id and returns the hash of
    // the transaction where the configuration was sent
    mapping (uint => bytes32[]) idToTxHashes;

    // Sends the configuration file for the specified VSB id
    function sendConfig(uint _id, string memory _data) public returns (uint){
        require( owner == msg.sender);
        return _id;
    }

    // Stores the hash of the transaction where the configuration was sent
    function broadcastTxHash(uint _id, bytes32 txHash) public returns (uint){
        require( owner == msg.sender);
        idToTxHashes[_id].push(txHash);
        return _id;
    }

    // Get the hash of the latest transaction where the config was sent for _id
    function getTxHash(uint _id) public view returns (bytes32) {
        uint lastElement = idToTxHashes[_id].length - 1;
        return idToTxHashes[_id][lastElement];
    }

    function getHashHistory(uint _id) public view returns (bytes32[] memory){
        return idToTxHashes[_id];
    }

}
```

Figura 9: Código fuente del contrato inteligente

Después de *sendConfig*, se define otra función, *broadcastTxHash*. Esta función es utilizada para almacenar en *idToTxHashes* el hash de la llamada a *sendConfig* ejecutada previamente, que contiene los datos de configuración cifrados para el nodo correspondiente. *broadcastTxHash* es ejecutada de forma pública. Así, cualquier nodo puede conocer los hashes de las transacciones en las que están guardados los datos de configuración para un determinado VSB, pero solo el nodo destinatario y el nodo autorizado pueden descifrar los contenidos de las transacciones de *sendConfig*.

Por otro lado, la función *getTxHash* recibe como entrada el ID del VSB para el cual se desea obtener la configuración más reciente, es decir, la configuración activa, y entrega como resultado el hash de la transacción correspondiente.

De esta forma, cuando un nodo necesita obtener su configuración, primero debe realizar una llamada a *getTxHash*, entregando como argumento su identificador. El resultado de esa llamada entregará un hash de transacción, y al revisar el contenido de esa transacción, se puede descifrar usando la clave privada que está almacenada en el gestor de transacciones privadas (Tessera).

Finalmente, se define la función *getHashHistory*, que, en lugar de devolver solo el hash de la última configuración, devuelve un arreglo de los hashes de todas las configuraciones que ha tenido un determinado VSB hasta ese momento.

6.3 Compilar el contrato inteligente

Una vez se ha definido el contrato, se necesita compilar el código fuente en Solidity a código que pueda ser ejecutado por la Ethereum Virtual Machine.

Para lograr esto, se utiliza la librería Web3.js. El código fuente que lleva a cabo la compilación se encuentra en el Anexo A, en el fichero */source/scripts/compile.js*

6.4 Desplegar el contrato inteligente

Una vez se tiene el código compilado, se debe desplegar en la *blockchain*.

Desplegar un contrato inteligente implica realizar una transacción que registra el contrato en la red y lo hace disponible para su interacción con otros nodos de la red.

Quorum permite el despliegue de contratos de forma privada, es decir, permitiendo que solo determinados nodos puedan ver e interactuar con el contrato. Sin embargo, en este caso, el despliegue se realiza de forma pública. En la Figura 10 se lista el código que despliega el contrato.

El *script* se conecta al nodo cero del entorno de pruebas, identificado por el puerto 22000. Luego, obtiene desde el sistema de archivos local el código compilado del contrato, desbloquea la cuenta asociada al nodo cero, y finalmente despliega el contrato e imprime en pantalla la información de la transacción, incluyendo la dirección del contrato. Además, guarda en el sistema de archivos local la dirección del contrato, ya que esta se debe conocer para poder interactuar con él.

En un entorno real, esta dirección sería posteriormente almacenada en cada nodo VSB, para que los nodos puedan interactuar con el contrato.

```
const fs = require('fs');
const Web3 = require('web3');

const node_url = 'http://localhost:22000';
const web3 = new Web3(node_url);

const abiFile = fs.readFileSync('../artifacts/abi.json');
const bytecodeFile = fs.readFileSync('../artifacts/bytecode.txt');

const abi = JSON.parse(abiFile);
const bytecode = bytecodeFile.toString();

const logDeployment = false;

const deployContract = async () => {
  try {
    await unlockAccount();
    const accounts = await web3.eth.getAccounts();
    const configSender = new web3.eth.Contract(abi);

    const deployment = await configSender.deploy({
      data: bytecode
    }).send({
      from: accounts[0],
      gas: '0x7b760'
      // No privateFor parameter: public deployment
    });

    console.log('Contrato desplegado! Dirección:', deployment.options.address);

    if (logDeployment){
      console.log(deployment);
    }

    fs.writeFile('./contract-address.txt', deployment.options.address, (err) => {
      if (err) {
        console.error('Error: no se pudo crear localmente el archivo de dirección de contrato.', err);
        return;
      }
      console.log('Dirección de contrato guardada en archivo ', 'contract-address.txt');
    });

  } catch (error) {
    console.log('Error al crear el contrato:', error);
  }
};

deployContract();

async function unlockAccount(){
  const password = '';
  const accounts = await web3.eth.getAccounts();
  const account = await web3.eth.personal.unlockAccount(accounts[0], password, null);
  console.log("Usando cuenta:", accounts[0]);
}
```

Figura 10: código de despliegue del contrato

6.5 Almacenar una configuración en la *blockchain*

Se desarrolló un script *sendConfig.js* encargado de enviar el archivo de configuración a la *blockchain* y luego publicar su hash para que sea visible a los nodos VSB.

El código íntegro del script se encuentra en el Anexo A, fichero *source/scripts/sendConfig.js*.

Al igual que en el despliegue, se obtiene desde el sistema de archivos local el código compilado del contrato, también conocido como ABI. Además, se obtiene la dirección del contrato ya desplegado.

A partir de esta información, se crea la variable *contract*, que sirve para interactuar con el contrato desplegado.

Luego, se pregunta al usuario por el ID del VSB que se desea configurar, la ruta del archivo que contiene la configuración deseada, y la clave pública del receptor.

La Figura 11 muestra la parte más importante de este script. En ella se aprecia cómo el código del script realiza directamente las dos llamadas a las funciones del contrato inteligente mencionadas anteriormente: *sendConfig* y *broadcastTxHash*.

La llamada a la función *sendConfig* del contrato es realizada entregando como argumento aquellos datos obtenidos del usuario en el paso previo. Pero, además, se incluye un objeto con los parámetros de transacción: *from*, *gas*, *gasPrice* y *PrivateFor*.

Los argumentos "from", "gas" y "gasPrice" son comunes en las transacciones de Ethereum y se utilizan para especificar lo siguiente:

1. "from": Este argumento indica la dirección de la cuenta desde la cual se está enviando la transacción. Es la dirección del remitente de la transacción y debe ser una cuenta válida en la red de Ethereum.

2. "gas": Este argumento define la cantidad de gas que se asignará a la transacción. El gas se utiliza para medir el costo computacional de ejecutar operaciones en la red de Ethereum. Cada operación consume una cierta cantidad de gas, y la cantidad total de gas utilizado afecta el costo de la transacción.

3. "gasPrice": Este argumento establece el precio del gas en wei (la unidad más pequeña de Ether). El precio del gas representa la tarifa que el remitente está dispuesto a pagar por cada unidad de gas utilizada en la transacción. Un gasPrice más alto generalmente resultará en que la transacción se procese más rápidamente, ya que los mineros tienen más incentivos para incluirla en un bloque.

Sin embargo, el parámetro *privateFor* es específico de Quorum, y es usado para designar a qué nodos se les permite acceder y ver los detalles de la transacción. En lugar de ser una transacción pública visible para todos en la red, una transacción privada solo es visible para los participantes autorizados.

Este parámetro recibe un arreglo que contiene las claves públicas de los nodos que deben tener acceso a los datos de la transacción. En este caso, se entrega solo una clave pública: la obtenida de la interacción con el usuario en el paso anterior.

Si la llamada a la función es exitosa, se muestra por pantalla el hash de la transacción.

Luego, este mismo hash es enviado como argumento para la función *broadcastTxHash*, que modifica el *mapping idToTxHashes*. Esta transacción no utiliza el parámetro *privateFor*, por lo que la transacción es pública: se permite a cualquier nodo conocer el hash de la transacción donde se envió la configuración, pero si el nodo no está autorizado, no podrá descifrar el contenido de la transacción de *sendConfig*.

Debido a este proceso, Grupo GTD debe manejar internamente una lista de las claves públicas de cada VSB.

```
const sendConfig = async (id, dataFileString, receiverPublicKey) => {

  const functionName = 'sendConfig';

  const _id = parseInt(id) || 0;
  const dataFile = fs.readFileSync(dataFileString);
  const data =  dataFile.toString();

  console.log("\n--sendConfig--");

  const _data = data || "";
  await unlockAccount();

  const functionParams = [_id, _data];

  const accounts = await web3.eth.getAccounts();

  await contract.methods[functionName](...functionParams).send({
      from: accounts[0],
      privateFor: [receiverPublicKey], // public keys stored in tessera node directory
      gas: '0x7b760',
      gasPrice: 0,
  }, (error, result) => {
    if (error) {
      console.error(error);
    } else {
      console.log("Tx Hash de sendConfig:", result);
      txHash = result;
    }
  });
}

const broadcastTxHash = async (id) => {

  console.log("\n--broadcastTxHash--");

  await unlockAccount();

  const _id = parseInt(id) || 0;
  const accounts = await web3.eth.getAccounts();

  const functionName = 'broadcastTxHash';
  const functionParams = [_id, txHash];

  await contract.methods[functionName](...functionParams).send({
      from: accounts[0],
  }, (error, result) => {
    if (error) {
      console.log("error")
      console.error(error);
    } else {
      console.log("hash de broadcastTxHash:", result);
    }
    console.log("Hash de Tx sendConfig difundido correctamente")
  });
}
```

Figura 11: interacción con funciones del contrato inteligente.

6.6 Obtener la configuración de un VSB

Cuando un VSB necesita obtener su configuración, debe realizar el proceso inverso: primero obtener el hash de la última transacción *sendConfig* que le corresponde y luego obtener la configuración.

El script que realiza esta función se encuentra en el archivo *autoConfig.js*, disponible en el Anexo A.

Este script toma como argumentos de consola o *Shell*:

1. El puerto del cliente Geth al que se debe conectar el script
2. El ID del VSB

Si el cliente Geth seleccionado posee en su gestor de transacciones privadas las claves correspondientes a las transacciones *sendConfig*, el script descarga automáticamente la configuración, y la almacena en un fichero en el sistema de archivos local.

Se recibe el puerto del nodo Geth como argumento de para poder facilitar el uso del script en el entorno de pruebas, verificando así la privacidad de las transacciones. Según el nodo que se seleccione, los datos de la transacción podrán ser descifrados o no.

La Figura 12 muestra las llamadas a funciones usando Web3.js.

La función *getTxHash* realiza una llamada a la función del mismo nombre en el contrato inteligente, obteniendo así el último hash de transacción que haya enviado una configuración para el ID entregado.

La función *getTransactiondata* utiliza el hash obtenido para obtener la transacción y descifrar los datos enviados en ella.

Cuando Web3.js solicita al nodo Geth obtener esta información, Geth toma los datos cifrados y delega la tarea de descifrarlos a su gestor de transacciones privadas, Tessera.

A su vez, el nodo Tessera utiliza su clave privada para descifrar el contenido de la transacción.

Si el contenido de la transacción no fue cifrado con la clave pública del nodo, el proceso falla.

6.7 Obtener el historial de configuraciones para un VSB

Además de la capacidad de obtener automáticamente la configuración activa o más reciente, la propuesta contempla la posibilidad de revisar el historial de configuraciones para un determinado VSB.

Este proceso se implementa utilizando dos scripts:

- *getHashHistory.js*: Obtiene los hashes de las transacciones *sendConfig* para un determinado ID.
- *getConfigFromHash.js*: Obtiene el archivo de configuración a partir de un determinado hash de transacción.

6.7.1 Obtener los hashes de las transacciones

En la Figura 13 se muestra el código fuente de *getHashHistory.js*.

La primera parte del script captura los argumentos de consola con los que este fue ejecutado. Solamente se recibe un argumento, que corresponde al ID del VSB para el cual se quiere conocer la historia.

Luego, se realiza una llamada a la función *getHashHistory*, definida en el contrato inteligente. Esta función retorna un arreglo con los hashes de las transacciones *sendConfig* que han sido enviadas para el VSB especificado.

6.7.2 Obtener la configuración a partir de un hash

En la Figura 14 se despliega el código fuente de *getTransactionFromHash.js*

Este script recibe dos argumentos:

1. Puerto del nodo Geth al que conectarse.
2. Hash de la transacción que se desea revisar.

A partir de esto, el script obtiene de la *blockchain* la transacción especificada, y luego despliega en pantalla el contenido descifrado de la transacción.

Como se puede apreciar, ambos scripts están diseñados para ser ejecutados manualmente por un ingeniero u operador del sistema, con el fin de diagnosticar errores y fallas de seguridad. No están diseñados para ser ejecutados automáticamente por los equipos VSB. Al ser el nodo autorizado el que envía las transacciones, este también es capaz de descifrarlas a partir del hash de cada transacción.

```
const fs = require('fs');
const Web3 = require('web3');
const Web3Quorum = require("web3js-quorum");
const abiDecoder = require('abi-decoder');

const node_url = 'http://localhost:' + port;

const web3 = new Web3Quorum(new Web3(node_url));

const abiFile = fs.readFileSync('../artifacts/abi.json');
const bytecodeFile = fs.readFileSync('../artifacts/bytecode.txt');
const contractAddressFile = fs.readFileSync('./contract-address.txt');
const contractAddress = contractAddressFile.toString();

const abi = JSON.parse(abiFile);
abiDecoder.addABI(abi);
const bytecode = bytecodeFile.toString();

const contract = new web3.eth.Contract(abi, contractAddress);

const getTxHash = async (id) => {

  const _id = parseInt(id) || 0;
  await unlockAccount();

  const accounts = await web3.eth.getAccounts();

  const functionName = 'getTxHash';
  const functionParams = [_id];

  contract.methods[functionName](...functionParams).call({
  }, (error, result) => {
    if (error) {
      console.log("error")
      console.error(error);
    } else {
      console.log("Hash de Tx sendConfig :", result);
      txHash = result;
    }
  });
}

const getTransactionData = async () => {
  await unlockAccount();

  transactionHash = txHash;

  // Get transaction
  var tx = await web3.eth.getTransaction(transactionHash);

  // Decrypt the transaction input field using the private keys of our node
  var decryptedInput = await web3.eth.getQuorumPayload(tx.input);

  // Decode the hexadecimal representation into the json data
  const decodedInput = await abiDecoder.decodeMethod(decryptedInput);

  console.log("Leyendo Tx:", transactionHash);

  console.log("contenido:", decodedInput.params[1].value);

  fs.writeFile('./config-file.txt', decodedInput.params[1].value, (err) => {
    if (err) {
      console.error('Error al guardar el archivo localmente', err);
      return;
    }
    console.log('archivo de configuración guardado en', 'config-file.txt');
  });
}
```

Figura 12: obtener configuración a partir de ID

```
let id_argument_fail = (!process.argv[2]) || (! Number(process.argv[2]) > 0 );

if (id_argument_fail) {
    console.log("Ejecutar como: ");
    console.log("$ node getHashHistory.js id");
    console.log("Donde `id` corresponde al identificador del VSB");
    console.log("Ejemplo:")
    console.log("$ node getHashHistory 24");
    process.exit();
}

const id = Number(process.argv[2]);

const fs = require('fs');
const Web3 = require('web3');
const Web3Quorum = require("web3js-quorum");
const abiDecoder = require('abi-decoder');

const node_url = 'http://localhost:22000';

const web3 = new Web3(node_url);

const abiFile = fs.readFileSync('../artifacts/abi.json');
const bytecodeFile = fs.readFileSync('../artifacts/bytecode.txt');
const contractAddressFile = fs.readFileSync('./contract-address.txt');
const contractAddress = contractAddressFile.toString();

const abi = JSON.parse(abiFile);
abiDecoder.addABI(abi);
const bytecode = bytecodeFile.toString();

const contract = new web3.eth.Contract(abi, contractAddress);

const getTxHash = async (id) => {

    const _id = parseInt(id) || 0;
    await unlockAccount();

    const accounts = await web3.eth.getAccounts();

    const functionName = 'getHashHistory';
    const functionParams = [_id];

    contract.methods[functionName](...functionParams).call({
    }, (error, result) => {
      if (error) {
        console.log("error")
        console.error(error);
      } else {
        console.log("Historial de hashes:", result);
      }
    });
}

async function main() {
    await getTxHash(id);
  }

main();

async function unlockAccount(){
    const password = '';
    const accounts = await web3.eth.getAccounts();
    const account = await web3.eth.personal.unlockAccount(accounts[0], password, null);
    console.log("Usando cuenta:", accounts[0]);
}
```

Figura 13: código para obtener historial

```
let port_condition_fail = (!process.argv[2]) || (! Number(process.argv[2]) > 0 )
let hash_argument_fail = (!process.argv[3]);

if (port_condition_fail || hash_argument_fail) {
    console.log("Ejecutar como: ");
    console.log("$ node getConfigFromHash.js puerto hash");
    console.log("donde `puerto` es el puerto para acceder al cliente geth");
    console.log("y `hash` corresponde al hash de la Tx que se desde revisar");
    console.log("Ejemplo:")
    console.log("$ node getConfigFromHash.js 22000 0xf6c...");
    process.exit();
}

let port = process.argv[2]
let txHash = process.argv[3]

const fs = require('fs');
const Web3 = require('web3');
const Web3Quorum = require("web3js-quorum");
const abiDecoder = require('abi-decoder');

const node_url = 'http://localhost:' + port;

const web3 = new Web3Quorum(new Web3(node_url));

const abiFile = fs.readFileSync('../artifacts/abi.json');
const bytecodeFile = fs.readFileSync('../artifacts/bytecode.txt');
const contractAddressFile = fs.readFileSync('./contract-address.txt');
const contractAddress = contractAddressFile.toString();

const abi = JSON.parse(abiFile);
abiDecoder.addABI(abi);
const bytecode = bytecodeFile.toString();

const contract = new web3.eth.Contract(abi, contractAddress);

const getTransactionData = async () => {
  await unlockAccount();

  transactionHash = txHash;

  // Get transaction
  var tx = await web3.eth.getTransaction(transactionHash);

  // Decrypt the transaction input field using the private keys of our node
  var decryptedInput = await web3.eth.getQuorumPayload(tx.input);

  // Decode the hexadecimal representation into the json data
  const decodedInput = await abiDecoder.decodeMethod(decryptedInput);

  console.log("Leyendo Tx:", transactionHash);

  console.log("contenido:", decodedInput.params[1].value);

  fs.writeFile('./config-file.txt', decodedInput.params[1].value, (err) => {
    if (err) {
      console.error('Error al guardar el archivo localmente', err);
      return;
    }
    console.log('archivo de configuración guardado en', 'config-file.txt');
  });
}

async function unlockAccount(){
    const password = '';
    const accounts = await web3.eth.getAccounts();
    const account = await web3.eth.personal.unlockAccount(accounts[0], password, null);
    console.log("Usando cuenta:", accounts[0]);
}

async function main() {
  await getTransactionData(txHash);
}

main();
```

Figura 14: Revisar una transacción específica

7. RESULTADOS

A continuación, se detallan los resultados del uso de la propuesta en el entorno de pruebas.

7.1 Despliegue del contrato

El *script* de compilación escanea el sistema de archivos local hasta encontrar el código fuente en Solidity, para luego compilarlo. Si la compilación es exitosa, se almacenan el *bytecode* y la ABI (código en formato ejecutable por la Ethereum Virtual Machine) como artefactos en el sistema de archivos local.

Al ejecutar el script *deploy.js*, se despliega el contrato usando los artefactos generados, y se muestra por pantalla la cuenta que se utilizó para el despliegue (que al desplegar el contrato se convierte en la cuenta *owner*), así como la dirección del contrato recién desplegado, tal como se ve en la Figura 15.

```
dasb@betelgeuse:~/Code/cc/source/scripts$ node compile.js
dasb@betelgeuse:~/Code/cc/source/scripts$ node deploy.js
Usando cuenta: 0x22b856bFf5129fEF80541201AA322D111c64Cc7A
Contrato desplegado! Dirección: 0xCdD1fe702BCFc64dCF019BB3502B26ceb50297bF
Dirección de contrato guardada en archivo  contract-address.txt
dasb@betelgeuse:~/Code/cc/source/scripts$
```

Figura 15: Compilación y despliegue del contrato

7.2 Envío de la configuración

El script *sendConfig.js* es interactivo. Pide al usuario entregar el ID del VSB que se desea configurar, la ruta al archivo local que contiene la configuración, y la clave pública del nodo destinatario, como se muestra en Figura 16.

```
dasb@betelgeuse:~/Code/cc/source/scripts$ node sendConfig.js
id del VSB: 14
Archivo a enviar: config-file.txt
Clave pública del destinatario:
```

Figura 16: *prompt* para enviar una configuración.

Al entregar todos los datos, el *script* realiza dos llamadas a funciones, tal como se vio en el capítulo anterior:

1. *SendConfig*, realizada como transacción privada.
2. *broadCastTxHash*, realizada de forma pública.

Si ambas transacciones son exitosas, se imprime por pantalla la cuenta utilizada para las transacciones, el hash de transacción de la llamada a *sendConfig*, y el hash de la llamada a broadcastTxHash, así como un mensaje de éxito. En la Figura 17, se muestra el resultado de la interacción, donde el usuario ingresa la clave pública del nodo dos. Posteriormente, este nodo podrá obtener la configuración del VSB con ID 14.

```
dasb@betelgeuse:~/Code/cc/source/scripts$ node sendConfig.js
id del VSB: 14
Archivo a enviar: config-file.txt
Clave pública del destinatario: OoyImxVb8A64KPdkCvg83GUdlXuf46IUvjlV7d5P4wk=

--sendConfig--
Usando cuenta: 0x22b856bFf5129fEF80541201AA322D111c64Cc7A
Tx Hash de sendConfig: 0x733daccef1fb7e64eac1b0bd7a21bb2c06d6f2ffd7a0fc021753c02
a282b70bd

--broadcastTxHash--
Usando cuenta: 0x22b856bFf5129fEF80541201AA322D111c64Cc7A
Tx Hash de broadcastTxhash: 0x6debdff76067583e0ac2e0fd57dc551dd9436ecd9ff16face7
ac95368730d50a
Hash de Tx sendConfig difundido correctamente
dasb@betelgeuse:~/Code/cc/source/scripts$
```

Figura 17: Enviar configuración especificando clave pública

7.3 Configuración automática del VSB

El *script autoConfig.js* recibe dos argumentos de terminal:

1. Puerto del cliente Geth.
2. ID del VSB.

Si el cliente corresponde con el ID, es decir, si el nodo posee la clave privada necesaria para descifrar el contenido de la transacción, entonces se descarga automáticamente la transacción y se instala, almacenando sus contenidos en un archivo local.

En la Figura 18 se muestra el resultado de ejecutar *autoConfig.js* en el nodo dos, para el ID de dispositivo 14.

```
dasb@betelgeuse:~/Code/cc/source/scripts$ node autoConfig.js 22002 14
Usando cuenta: 0xC2d885B03D65EB0AaD695A43AaB3868B6c01E1bc
Hash de Tx sendConfig : 0x733daccef1fb7e64eac1b0bd7a21bb2c06d6f2ffd7a0fc021753c0
2a282b70bd
Usando cuenta: 0xC2d885B03D65EB0AaD695A43AaB3868B6c01E1bc
Leyendo Tx: 0x733daccef1fb7e64eac1b0bd7a21bb2c06d6f2ffd7a0fc021753c02a282b70bd
contenido: {
    vsb_id= 14,
    variable1: "value1",
    variable2: "value2",
    anotherVariable: "anotherValue"
}


archivo de configuración guardado en config-file.txt
dasb@betelgeuse:~/Code/cc/source/scripts$
```

Figura 18: Configuración automática del nodo

Si el cliente Geth especificado no posee las claves públicas para descifrar el contenido de la transacción, el resultado es un mensaje de error.

En la Figura 19, se muestra una llamada a *autoConfig.js* donde el script se conecta al nodo uno (puerto 22001). Como no se ha enviado ninguna configuración para el nodo 14 utilizando la clave pública del nodo uno, la configuración automática falla.

```
dasb@betelgeuse:~/Code/cc/source/scripts$ node autoConfig.js 22001 14
Usando cuenta: 0x3151d213EC57F263B1AeE0a1c21BF47b298daD92
Hash de Tx sendConfig : 0xed24da444e005e107b96f837bd4aad439f50f78c9b7b20550d3c92
4ca604e7b1
Usando cuenta: 0x3151d213EC57F263B1AeE0a1c21BF47b298daD92
Leyendo Tx: 0xed24da444e005e107b96f837bd4aad439f50f78c9b7b20550d3c924ca604e7b1
/home/dasb/Code/cc/source/scripts/autoConfig.js:75
  console.log("contenido:", decodedInput.params[1].value);
                                          ^

TypeError: Cannot read properties of undefined (reading 'params')
    at getTransactionData (/home/dasb/Code/cc/source/scripts/autoConfig.js:75:42
)
    at processTicksAndRejections (node:internal/process/task_queues:96:5)
    at async main (/home/dasb/Code/cc/source/scripts/autoConfig.js:96:3)
dasb@betelgeuse:~/Code/cc/source/scripts$
```

Figura 19: Instalación fallida al intentar obtener configuración ajena

En un entorno real, se recomienda configurar previamente los VSB para ejecutar periódicamente este script con los argumentos correspondientes usando la herramienta de *crontab*, utilizada en entornos UNIX, en el propio VSB, que permite ejecutar programas de forma periódica sin intervención del usuario. De esta forma, se instala automáticamente la configuración una vez esta es actualizada por el nodo autorizado.

7.4 Revisión manual de configuración

Además, es posible para el operador o ingeniero de la empresa revisar el historial de configuraciones de forma manual, utilizando los scripts *getHashHistory.js* y *getConfigFromHash.js*, como se muestra en la Figura 20.

Primero, se ejecuta *getHashHistory.js* entregando como argumento de consola el ID del VSB para el cual se desea revisar la configuración. Como respuesta, se muestra un arreglo que contiene los hashes de las transacciones sendConfig para ese VSB.

Luego, se ejecuta *getConfigFromHash.js* que recibe como argumentos el puerto del cliente Geth, así como el hash que se desea revisar.

El script utiliza la cuenta y el nodo especificados para revisar la transacción e intentar descifrar sus contenidos. Si el nodo es el correcto, se imprime por pantalla el contenido del archivo de configuración.

De esta forma, al poder ver las configuraciones pasadas de un VSB, se facilita el diagnóstico de problemas y la recuperación ante posibles eventos catastróficos o inesperados.

```
dasb@betelgeuse:~/Code/cc/source/scripts$ node getHashHistory.js 14
Usando cuenta: 0x22b856bFf5129fEF80541201AA322D111c64Cc7A
Historial de hashes: [
  '0x733daccef1fb7e64eac1b0bd7a21bb2c06d6f2ffd7a0fc021753c02a282b70bd'
]
dasb@betelgeuse:~/Code/cc/source/scripts$ node getConfigFromHash.js 22002 0x733d
accef1fb7e64eac1b0bd7a21bb2c06d6f2ffd7a0fc021753c02a282b70bd
Usando cuenta: 0xC2d885B03D65EB0AaD695A43AaB3868B6c01E1bc
Leyendo Tx: 0x733daccef1fb7e64eac1b0bd7a21bb2c06d6f2ffd7a0fc021753c02a282b70bd
contenido: {
    vsb_id= 14,
    variable1: "value1",
    variable2: "value2",
    anotherVariable: "anotherValue"
}



archivo de configuración guardado en config-file.txt
dasb@betelgeuse:~/Code/cc/source/scripts$
```

Figura 20: Revisar historial manualmente

8. CUMPLIMIENTO DE OBJETIVOS

8.1 Objetivos Específicos

8.1.1. Diseñar un sistema de sincronización que mediante una red distribuida permita a los VSB obtener configuraciones al estar en la red del cliente.

Se desarrolló un sistema de sincronización que, mediante una red *blockchain* descentralizada, permite que los VSB automáticamente descarguen sus configuraciones, tal como se observa en el capítulo de resultados.

Esto es posible en cualquier lugar donde el equipo VSB tenga acceso a la red de Grupo GTD, incluyendo al estar en la red del cliente.

8.1.2. Automatizar el proceso de sincronización de configuración mediante software que se ejecuta en el VSB

Tal como se mencionó en el capítulo de resultados, es posible automatizar el proceso de descarga e instalación de configuración, ya que el script *autoConfig.js* no requiere interacción del usuario, solo requiere argumentos de consola que pueden configurarse previamente para su ejecución automática en el equipo en intervalos periódicos.

8.1.3. Mejorar la persistencia de las configuraciones mediante una arquitectura distribuida

En el Marco Teórico se explicó cómo la tecnología *blockchain* asegura la integridad de la información, impidiendo la manipulación de los datos contenidos en la *blockchain* por parte de posibles actores maliciosos.

El contrato inteligente desplegado especifica formas de agregar información a la *blockchain*, mediante las funciones *sendConfig* y *broadcastTxHash*, actualizando así la configuración activa en cada momento para un determinado VSB, pero no se permite eliminar esta información, ni modificar archivos enviados anteriormente.

De esta forma, como se explica en el capítulo de implementación y se muestra en el capítulo de resultados, es posible para el nodo autorizado revisar el historial de configuraciones de un VSB a partir de su ID.

Además, el sistema implementado realiza una replicación de la información en cada nodo. La información está cifrada usando las claves públicas del nodo autorizado y del nodo VSB destinatario, por lo que cualquiera de estos dos nodos tiene la capacidad de revisar el historial de configuraciones en cualquier momento.

Esto implica que se supera el “punto central de falla” que tienen los modelos centralizados. Además, al almacenar el historial de configuraciones, hay una mejora en la persistencia respecto al sistema de configuraciones actual.

8.1.4. Evaluar el funcionamiento del sistema y su aplicabilidad en el contexto de un CSP

El sistema desarrollado cumple los requisitos de usuario definidos.

Por medio del entorno de pruebas, se demuestra que los equipos pueden comunicarse exitosamente en una red *blockchain* descentralizada, y el software se ejecuta en computadores de propósito general, tal como los VSB.

Adicionalmente, se garantiza tanto la privacidad de las transacciones, utilizando la funcionalidad de transacciones privadas de Quorum, como la integridad de los datos almacenados en la *blockchain*, gracias a las propiedades explicadas en el Marco Teórico.

La red de un CSP es, además, un entorno apropiado para una red *blockchain* privada, y cumple las características descritas en el estado del arte para este tipo de red.

8.2 Objetivo General

Se definió como objetivo general el “Evaluar una arquitectura distribuida que permita la sincronización de información entre nodos que formen parte de un ambiente Edge Computing”.

La propuesta desarrollada demuestra la factibilidad técnica de utilizar una aplicación descentralizada en una red *blockchain* como plataforma de distribución y sincronización de información en el ambiente señalado.

Por lo tanto, visto el cumplimiento de los objetivos secundarios, más la factibilidad técnica mencionada, se da por cumplido el objetivo general.

9. CONCLUSIONES

A raíz de la revisión sistemática desarrollada, se concluyó, por un lado, que las redes *blockchain* privadas son viables en la práctica y, por otro lado, que es posible realizar transacciones confidenciales entre dos nodos de la red sin que el resto de los nodos tengan acceso a los contenidos de la transacción.

A partir de estos descubrimientos, se definió una propuesta para utilizar una *blockchain* privada para lograr sincronizar datos en un entorno de un CSP (*Communication Service Provider*) utilizando Quorum, una variante de Ethereum que hace posibles las transacciones privadas entre nodos de la red utilizando criptografía de clave pública.

La implementación del sistema y los resultados demuestran la factibilidad técnica de la propuesta. Se implementó un sistema que exitosamente permite a un nodo autorizado publicar archivos de configuración para que cada uno de los nodos VSB de la red pueda descargar e instalar su archivo de forma automática.

Mediante esta propuesta, se mejora la trazabilidad de las configuraciones, porque se permite al operador de la red revisar el historial de transacciones que ha tenido cada VSB a lo largo del tiempo. El historial de transacciones pasadas es inmutable y asegura la integridad de los datos, gracias al uso de *blockchain*.

Además, se detallan algunos procesos que debe realizar Grupo GTD para posibilitar el funcionamiento del sistema propuesto, que incluyen: instalar en los VSB el software necesario para ejecutar Geth, el cliente de Quorum modificado; y Tessera, el gestor de transacciones privadas; un archivo que identifque la ID del VSB, así como la mantención de una lista de correspondencia entre el ID y la clave pública de cada nodo.

El esfuerzo requerido en estos procesos de negocio necesarios para el funcionamiento de la propuesta podría ser considerable. Este trabajo no pretende indicar a la empresa si la propuesta es mejor que las alternativas centralizadas, sino que se intenta reflejar las ventajas que entrega el uso de *blockchain* para la sincronización de datos.

10. BIBLIOGRAFÍA

Ante, L., & Fiedler, I. (2021). Bitcoin's energy consumption and social costs in relation to its capacity as a settlement layer.

Anwar, A. (2021). *What is a Private Blockchain? Beginner's Guide*. Obtenido de 101 Blockchains: https://101blockchains.com/what-is-a-private-blockchain/

Aponte-Novoa, F., Orozco, A. L., Villanueva-Polanco, R., & Wightman, P. (2021). The 51% Attack on Blockchains: A Mining Behavior Study. *IEEE Access*.

Back, A. (1997). *[ANNOUNCE] hash cash postage implementation*. Obtenido de http://www.hashcash.org/papers/announce.txt

Back, A. (2002). Hashcash - A Denial of Service Counter-Measure.

Baliga, A., Subhod, i., Kamat, P., & Chatterjee, S. (2018). *Performance Evaluation of the Quorum Blockchain Platform*. Obtenido de https://arxiv.org/pdf/1809.03421.pdf#cite.raft

Benet, J. (2019). *IPFS - Content Addressed, Versioned, P2P File System*. Obtenido de https://arxiv.org/pdf/1407.3561.pdf

Bertolini, P. (2 de Diciembre de 2021). *GTD inaugura el cable submarino Prat para impulsar conectividad en Chile*. Obtenido de DPL News: https://dplnews.com/gtd-inaugura-el-cable-submarino-prat-para-impulsar-conectividad-en-chile/

Boetticher, J. (2023). *How to Build a DApp: Complete DApp Architecture*.

Brakeville, S., & Perepa, B. (2019). *Blockchain basics: Introduction to distributed ledgers*. Obtenido de https://developer.ibm.com/tutorials/cl-blockchain-basics-intro-bluemix-trs/

Buterin, V. (2014). *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.* Obtenido de Ethereum.org: https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf

Buterin, V. (2020). Obtenido de https://vitalik.ca/general/2020/11/06/pos2020.html

Cant, J. (2019). *Monero Implements Hard Fork, Including New ASIC-Resistant Mining Algorithm* . Obtenido de https://cointelegraph.com/news/monero-implements-hard-fork-including-new-asic-resistant-mining-algorithm

Chen, W., Xu, Z., Shi, S., Zhao, Y., & Zhao, J. (2019). *A Survey of Blockchain Applications in Different Domains*. Obtenido de https://arxiv.org/ftp/arxiv/papers/1911/1911.02013.pdf

Chohan, U. W. (2021). The Double Spending Problem and Cryptocurrencies.

CloudFlare. (s.f.). *What is TLS (Transport Layer Security)?* Obtenido de CloudFlare: https://www.cloudflare.com/en-gb/learning/ssl/transport-layer-security-tls/

Cohen, B. (2002). *The BitTorrent Protocol Specification*. Obtenido de https://www.bittorrent.org/beps/bep_0003.html

Consensys. (2018). *Scaling Consensus for Enterprise: Explaining the IBFT Algorithm*. Obtenido de https://consensys.net/blog/enterprise-blockchain/scaling-consensus-for-enterprise-explaining-the-ibft-algorithm/

Consensys Quorum. (2023). *Consensys Quorum*. Obtenido de https://consensys.net/quorum/

Corda. (s.f.). Obtenido de https://corda.net/

Crypto Council for Innovation. (2022). *Real-World Use Cases for Smart Contracts and dApps*. Obtenido de Crypto Council for Innovation: https://cryptoforinnovation.org/real-world-use-cases-for-smart-contracts-and-dapps/

Crypto.com. (2022). *What Are Token Standards? An Overview*. Obtenido de https://crypto.com/university/what-are-token-standards

Dai, W. (1998). Obtenido de http://www.weidai.com/bmoney.txt

de Isidro, R., Anderson, E., & Reddy, R. (2022). *Proof of Work vs. Proof of Stake: Why Their Differences Matter*. Obtenido de https://www.globalxetfs.com/proof-of-work-vs-proof-of-stake-why-their-differences-matter/

Driscoll, K., Hall, B., Sivencrona, H., & Zumsteg, P. (2003). *Byzantine Fault Tolerance, from Theory to Reality*.

Dwork, C., & Naor, M. (1992). Pricing via Processing or Combatting Junk Mail.

Ethereum.org. (2023). *Ethereum Glossary*. Obtenido de Ethereum.org: https://ethereum.org/en/glossary/#solidity

Ethereum.org. (2023). *The Merge*. Obtenido de Ethereum.org: https://ethereum.org/en/roadmap/merge/

getmonero.org. (s.f.). *Mining Monero*. Obtenido de https://www.getmonero.org/get-started/mining/

Grupo GTD. (2022). *Cable Submarino de Fibra Óptica Prat*. Obtenido de Grupo GTD: https://www.gtd.cl/cable-submarino-de-fibra-optica

Grupo GTD. (2023a). *Soluciones Digitales, Conectividad y Seguridad para tu Empresa - GTD Chile - GTD*. Obtenido de GTD Chile: https://gtd.cl/empresas

Grupo GTD. (2023b). *Nuestra Empresa*. Obtenido de Grupo GTD: https://www.gtd.cl/nuestra-empresa/grupo-gtd

Grupo GTD. (2023c). *housing data center*. Obtenido de Grupo GTD: https://www.gtd.cl/empresas/soluciones/cloud/servicios-de-data-center/housing-puerto-montt

Grupo GTD. (2023c). *Objetivo Corporativo*. Obtenido de Grupo GTD: https://www.gtd.cl/nuestra-empresa/objetivo-corporativo

Grupo GTD. (2023d). *Gtd y Starlink: Mayor robustez y resiliencia para tu empresa*. Obtenido de https://www.gtd.cl/es/w/novedades/gtd-y-starlink-mayor-robustez-y-resiliencia-para-su-empresa

Grupo GTD. (2023e). *Información Corporativa*. Obtenido de https://www.gtd.cl/nuestra-empresa/informacion-corporativa

Gul, M., Rehman, A., & Paul, A. e. (2020). *Blockchain Expansion to secure Assets with Fog Node on special Duty*. Obtenido de https://doi.org/10.1007/s00500-020-04857-0

HashCash. (s.f.). *Hashcash FAQ*. Obtenido de http://www.hashcash.org/faq/

Hay, T. (2021). Obtenido de https://consensys.net/blog/quorum/hyperledger-besu-understanding-proof-of-authority-via-clique-and-ibft-2-0-private-networks-part-1/

Hellman, M. E. (1978). *An Overview of Public Key Cryptography*. Obtenido de https://www-ee.stanford.edu/~hellman/publications/31.pdf

Hyperledger. (2019). *How Walmart brought unprecedented transparency to the food supply chain with Hyperledger Fabric*. Obtenido de https://www.hyperledger.org/wp-content/uploads/2019/02/Hyperledger_CaseStudy_Walmart_Printable_V4.pdf

HyperLedger. (2022). *Hyperldger Fabric*. Obtenido de https://www.hyperledger.org/use/fabric

Kashyap, B. (2023). *Proof-of-stake (PoS)*. Obtenido de Ethereum.org: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/

Katz, J., & Lindell, Y. (2007). *Introduction to Modern Cryptography*.

Kleine, D. (2019). *Cryptocurrency Mining: Are ASICs Causing Centralization?* Obtenido de Crypto Briefing: https://cryptobriefing.com/cryptocurrency-mining-asic-centralization/

koonopek. (2023). *Gas and fees*. Obtenido de Ethereum.org: https://ethereum.org/en/developers/docs/gas/

Košťál, K., Helebrandt, P., Belluš, M., Ries, M., & Kotuliak, I. (2019). *Management and Monitoring of IoT Devices Using Blockchain* .

La Tercera. (2010). *Icare premia a GTD y dueño de Multihogar* . Obtenido de La Tercera: https://www.latercera.com/diario-impreso/icare-premia-a-gtd-y-dueno-de-multihogar/

Lamport, L., Shostak, R., & Pease, M. (2016). *The Byzantine Generals Problem*. Obtenido de https://www.microsoft.com/en-us/research/uploads/prod/2016/12/The-Byzantine-Generals-Problem.pdf

Leal, J. (2023). *What are EVM Compatible Blockchains? A Guide to the Ethereum Virtual Machine*. Obtenido de Third Web: https://blog.thirdweb.com/evm-compatible-blockchains-and-ethereum-virtual-machine/

Ledmi, A., Bendjenna, H., & Hemam, S. M. (2018). *Fault Tolerance in Distributed Systems: A Survey.*

Mohapatra, Bhoi, Jena, Nayak, & Singh. (2022). *A blockchain security scheme to support fog-based internet of things* .

Moore, G. A. (1991). *Crossing the Chasm: marketing and selling disruptive products to mainstream customers.*

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

Nguyen, C. T., Hoang, D. T., Nguyen, D., Niyato, D., Nguyen, H. T., & Dutkiewicz, E. (2019). Proof-of-Stake Consensus Mechanisms for Future.

Nguyen, K. T., Babar, M. A., & Boan, J. (2021).

Nguyen, K. T., Babar, M. A., & Boan, J. (2021). *). Integrating blockchain and Internet of Things systems: A systematic review on objectives and designs*.

Oktian, L. (2021). *BorderChain: Blockchain-Based Access Control Framework for the Internet of Things Endpoint* .

Pasick, A. (16 de Noviembre de 2004). *LIVEWIRE - File-sharing network thrives beneath the radar*. Obtenido de https://web.archive.org/web/20050527212007/http://in.tech.yahoo.com/041103/137/2ho4i.html

Pathirana, A., & Halgamuge, M. (2019). Energy efficient bitcoin mining to maximize the mining profit: Using data from 119 bitcoin mining hardware setups.

Patron, T. (2016). *What's the Big Idea Behind Ethereum's World Computer?* Obtenido de Coindesk: https://www.coindesk.com/markets/2016/03/13/whats-the-big-idea-behind-ethereums-world-computer/

Peaster, W. M. (2020). *Ethereum Gas Explained*. Obtenido de DeFiPrime: https://defiprime.com/gas

Smith, C. (2023a). *Introduction to dapps*. Obtenido de Ethereum.org: https://ethereum.org/en/developers/docs/dapps/

Smith, C. (2023b). *Intro to Ether*. Obtenido de Ethereum.org: https://ethereum.org/en/developers/docs/intro-to-ether/

Sristy, A. (2021). *Blockchain in the food supply chain - What does the future look like?* Obtenido de https://tech.walmart.com/content/walmart-global-tech/en_us/news/articles/blockchain-in-the-food-supply-chain.html

Tanenbaum, A., & van Steen, M. (1996). *Distributed Systems: Concepts and Design.*

The Solidity Authors. (2023). *Language Influences*. Obtenido de Solidity Documentation: https://docs.soliditylang.org/en/latest/language-influences.html

Wackerow, P. (2022). *Smart contract languages*. Obtenido de Ethereum.org: https://ethereum.org/en/developers/docs/smart-contracts/languages/#solidity

Wackerow, P. (2022a). *Introduction to smart contracts*. Obtenido de Ethereum.org: https://ethereum.org/en/developers/docs/smart-contracts/

Wackerow, P. (2022b). *Smart contract languages*. Obtenido de https://ethereum.org/en/developers/docs/smart-contracts/languages/#solidity

Wood, G. (2023). ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER. Obtenido de Ethereum.org: https://ethereum.org/en/developers/docs/evm/

Wu, Dong, Ota, Li, & Yang. (2020). *Application-Aware Consensus Management for Software-Defined Intelligent Blockchain in IoT*.

11. ANEXO A: CÓDIGO FUENTE

11.1 source/contract/configSender.sol

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.5.13.

contract configSender{

    //node authorized to publish blocks
    address private owner;

    constructor() public{
        owner = msg.sender;
    }

    // Takes the VSB id and returns the hash of
    // the transaction where the configuration was sent
    mapping (uint => bytes32[]) idToTxHashes;

    // Sends the configuration file for the specified VSB id
    function sendConfig(uint _id, string memory _data) public returns (uint){
        require( owner == msg.sender);
        return _id;
    }

    // Stores the hash of the transaction where the configuration was sent
    function broadcastTxHash(uint _id, bytes32 txHash) public returns (uint){
        require( owner == msg.sender);
        idToTxHashes[_id].push(txHash);
        return _id;
    }

    // Get the hash of the latest transaction where the config was sent for _id
    function getTxHash(uint _id) public view returns (bytes32) {
        uint lastElement = idToTxHashes[_id].length - 1;
        return idToTxHashes[_id][lastElement];
```

```
    }

    function getHashHistory(uint _id) public view returns (bytes32[] memory){
        return idToTxHashes[_id];
    }

}
```

11.2 source/scripts/compile.js

```
const solc = require('solc');
const fs = require('fs');

// Get contract source code
const contractPath = '../contracts/configSender.sol';
const contractSource = fs.readFileSync(contractPath, 'utf8');

const input = {
  language: 'Solidity',
  sources: {
    'configSender.sol': {
      content: contractSource
    }
  },
  settings: {
    outputSelection: {
      '*': {
        '*': ['*']
      }
    }
  }
};

// Compile
const output = JSON.parse(solc.compile(JSON.stringify(input)));

// Get the compiled contract bytecode and ABI
```

```
const contractName = 'configSender';

const contractBytecode =
output.contracts['configSender.sol'][contractName].evm.bytecode.object;

const contractAbi = output.contracts['configSender.sol'][contractName].abi;


// Save the bytecode and abi in the artifacts directory

const write1 = fs.writeFileSync('../artifacts/bytecode.txt', contractBytecode);

const write2 = fs.writeFileSync('../artifacts/abi.json', JSON.stringify(contractAbi));
```

11.3 source/scripts/deploy.js

```
const fs = require('fs');

const Web3 = require('web3');


const node_url = 'http://localhost:22000';

const web3 = new Web3(node_url);


const abiFile = fs.readFileSync('../artifacts/abi.json');

const bytecodeFile = fs.readFileSync('../artifacts/bytecode.txt');


const abi = JSON.parse(abiFile);

const bytecode = bytecodeFile.toString();


const logDeployment = false;


const deployContract = async () => {

  try {

    await unlockAccount();

    const accounts = await web3.eth.getAccounts();

    const configSender = new web3.eth.Contract(abi);


    const deployment = await configSender.deploy({

      data: bytecode

    }).send({

      from: accounts[0],

      gas: '0x7b760'

      // No privateFor parameter: public deployment

    });
```

```
    console.log('Contrato desplegado! Dirección:', deployment.options.address);


    if (logDeployment){
      console.log(deployment);
    }


    fs.writeFile('./contract-address.txt', deployment.options.address, (err) => {
      if (err) {
        console.error('Error: no se pudo crear localmente el archivo de dirección de
contrato.', err);
        return;
      }
      console.log('Dirección de contrato guardada en archivo ', 'contract-address.txt');
    });


  } catch (error) {
    console.log('Error al crear el contrato:', error);
  }
};


deployContract();


async function unlockAccount(){
  const password = '';
  const accounts = await web3.eth.getAccounts();
  const account = await web3.eth.personal.unlockAccount(accounts[0], password, null);
  console.log("Usando cuenta:", accounts[0]);
}
```

11.4 source/scripts/sendConfig.js

```
const fs = require('fs');
const Web3 = require('web3');
const readline = require('readline');


const node_url = 'http://localhost:22000';
const web3 = new Web3(node_url);
```

```
const abiFile = fs.readFileSync('../artifacts/abi.json');
const bytecodeFile = fs.readFileSync('../artifacts/bytecode.txt');
const contractAddressFile = fs.readFileSync('./contract-address.txt');
const contractAddress = contractAddressFile.toString();

const abi = JSON.parse(abiFile);
const bytecode = bytecodeFile.toString();

const contract = new web3.eth.Contract(abi, contractAddress);

var txHash = "";

async function main(){
  const [id, dataFileString, receiverPublicKey] = await promptUser();
  await sendConfig(id, dataFileString, receiverPublicKey);
  await broadcastTxHash(id);
}

main();

const sendConfig = async (id, dataFileString, receiverPublicKey) => {

  const functionName = 'sendConfig';

  const _id = parseInt(id) || 0;
  const dataFile = fs.readFileSync(dataFileString);
  const data =  dataFile.toString();

  console.log("\n--sendConfig--");

  const _data = data || "";
  await unlockAccount();

  const functionParams = [_id, _data];
```

```
  const accounts = await web3.eth.getAccounts();

  await contract.methods[functionName](...functionParams).send({
      from: accounts[0],
      privateFor: [receiverPublicKey], // public keys stored in tessera node directory
      gas: '0x7b760',
      gasPrice: 0,
  }, (error, result) => {
    if (error) {
      console.error(error);
    } else {
      console.log("Tx Hash de sendConfig:", result);
      txHash = result;
    }
  });
}

const broadcastTxHash = async (id) => {

  console.log("\n--broadcastTxHash--");

  await unlockAccount();

  const _id = parseInt(id) || 0;
  const accounts = await web3.eth.getAccounts();

  const functionName = 'broadcastTxHash';
  const functionParams = [_id, txHash];

  await contract.methods[functionName](...functionParams).send({
      from: accounts[0],
  }, (error, result) => {
    if (error) {
      console.log("error")
      console.error(error);
    } else {
```

```
      console.log("Tx Hash de broadcastTxhash:", result);
    }
    console.log("Hash de Tx sendConfig difundido correctamente")
  });
}

async function askQuestion(question) {
  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
  });

  return new Promise((resolve) => {
    rl.question(question, (answer) => {
      rl.close();
      resolve(answer);
    });
  });
}

async function promptUser() {
  const id = await askQuestion('id del VSB: ');
  const dataFileString = await askQuestion('Archivo a enviar: ');
  const receiverPublicKey = await askQuestion("Clave pública del destinatario: ");
  return [id, dataFileString, receiverPublicKey];
}

async function unlockAccount(){
  const password = '';
  const accounts = await web3.eth.getAccounts();
  const account = await web3.eth.personal.unlockAccount(accounts[0], password, null);
  console.log("Usando cuenta:", accounts[0]);
}
```

11.5 source/scripts/autoConfig.js

```
let port_condition_fail = (!process.argv[2]) || (! Number(process.argv[2]) > 0 )
let id_argument_fail = (!process.argv[3]) || (! Number(process.argv[3]) > 0 );

if (port_condition_fail || id_argument_fail) {
    console.log("Ejecutar como: ");
    console.log("$ node autoConfig.js puerto id");
    console.log("donde `puerto` es el puerto para acceder al cliente geth");
    console.log("e `id` corresponde al identificador del VSB");
    console.log("Ejemplo:")
    console.log("$ node autoConfig.js 22000 24");
    process.exit();
}

port = process.argv[2]
id = Number(process.argv[3]);

const fs = require('fs');
const Web3 = require('web3');
const Web3Quorum = require("web3js-quorum");
const abiDecoder = require('abi-decoder');

const node_url = 'http://localhost:' + port;

const web3 = new Web3Quorum(new Web3(node_url));

const abiFile = fs.readFileSync('../artifacts/abi.json');
const bytecodeFile = fs.readFileSync('../artifacts/bytecode.txt');
const contractAddressFile = fs.readFileSync('./contract-address.txt');
const contractAddress = contractAddressFile.toString();

const abi = JSON.parse(abiFile);
abiDecoder.addABI(abi);
const bytecode = bytecodeFile.toString();

const contract = new web3.eth.Contract(abi, contractAddress);
```

```
const getTxHash = async (id) => {

  const _id = parseInt(id) || 0;
  await unlockAccount();

  const accounts = await web3.eth.getAccounts();

  const functionName = 'getTxHash';
  const functionParams = [_id];

  contract.methods[functionName](...functionParams).call({
  }, (error, result) => {
    if (error) {
      console.log("error")
      console.error(error);
    } else {
      console.log("Hash de Tx sendConfig :", result);
      txHash = result;
    }
  });
}

const getTransactionData = async () => {
  await unlockAccount();

  transactionHash = txHash;

  // Get transaction
  var tx = await web3.eth.getTransaction(transactionHash);

  // Decrypt the transaction input field using the private keys of our node
  var decryptedInput = await web3.eth.getQuorumPayload(tx.input);

  // Decode the hexadecimal representation into the json data
  const decodedInput = await abiDecoder.decodeMethod(decryptedInput);
```

```
  console.log("Leyendo Tx:", transactionHash);

  console.log("contenido:", decodedInput.params[1].value);

  fs.writeFile('./config-file.txt', decodedInput.params[1].value, (err) => {
    if (err) {
      console.error('Error al guardar el archivo localmente', err);
      return;
    }
    console.log('archivo de configuración guardado en', 'config-file.txt');
  });
}

async function unlockAccount(){
    const password = '';
    const accounts = await web3.eth.getAccounts();
    const account = await web3.eth.personal.unlockAccount(accounts[0], password, null);
    console.log("Usando cuenta:", accounts[0]);
}

async function main() {
  var txHash;
  await getTxHash(id);
  await getTransactionData(txHash);
}

main();
```

11.6 source/scripts/getHashHistory.js

```
let id_argument_fail = (!process.argv[2]) || (! Number(process.argv[2]) > 0 );

if (id_argument_fail) {
    console.log("Ejecutar como: ");
```

```
    console.log("$ node getHashHistory.js id");
    console.log("Donde `id` corresponde al identificador del VSB");
    console.log("Ejemplo:")
    console.log("$ node getHashHistory 24");
    process.exit();
}

const id = Number(process.argv[2]);

const fs = require('fs');
const Web3 = require('web3');
const Web3Quorum = require("web3js-quorum");
const abiDecoder = require('abi-decoder');

const node_url = 'http://localhost:22000';

const web3 = new Web3(node_url);

const abiFile = fs.readFileSync('../artifacts/abi.json');
const bytecodeFile = fs.readFileSync('../artifacts/bytecode.txt');
const contractAddressFile = fs.readFileSync('./contract-address.txt');
const contractAddress = contractAddressFile.toString();

const abi = JSON.parse(abiFile);
abiDecoder.addABI(abi);
const bytecode = bytecodeFile.toString();

const contract = new web3.eth.Contract(abi, contractAddress);

const getTxHash = async (id) => {

    const _id = parseInt(id) || 0;
    await unlockAccount();

    const accounts = await web3.eth.getAccounts();
```

```
    const functionName = 'getHashHistory';
    const functionParams = [_id];

    contract.methods[functionName](...functionParams).call({
    }, (error, result) => {
      if (error) {
        console.log("error")
        console.error(error);
      } else {
        console.log("Historial de hashes:", result);
      }
    });
}

async function main() {
    await getTxHash(id);
  }

main();

async function unlockAccount(){
    const password = '';
    const accounts = await web3.eth.getAccounts();
    const account = await web3.eth.personal.unlockAccount(accounts[0], password, null);
    console.log("Usando cuenta:", accounts[0]);
}
```

11.7 source/scripts/getConfigFromHash.js

```
let port_condition_fail = (!process.argv[2]) || (! Number(process.argv[2]) > 0 )
let hash_argument_fail = (!process.argv[3]);

if (port_condition_fail || hash_argument_fail) {
    console.log("Ejecutar como: ");
    console.log("$ node getConfigFromHash.js puerto hash");
```

```
    console.log("donde `puerto` es el puerto para acceder al cliente geth");
    console.log("y `hash` corresponde al hash de la Tx que se desde revisar");
    console.log("Ejemplo:")
    console.log("$ node getConfigFromHash.js 22000 0xf6c...");
    process.exit();
}

let port = process.argv[2]
let txHash = process.argv[3]

const fs = require('fs');
const Web3 = require('web3');
const Web3Quorum = require("web3js-quorum");
const abiDecoder = require('abi-decoder');

const node_url = 'http://localhost:' + port;

const web3 = new Web3Quorum(new Web3(node_url));

const abiFile = fs.readFileSync('../artifacts/abi.json');
const bytecodeFile = fs.readFileSync('../artifacts/bytecode.txt');
const contractAddressFile = fs.readFileSync('./contract-address.txt');
const contractAddress = contractAddressFile.toString();

const abi = JSON.parse(abiFile);
abiDecoder.addABI(abi);
const bytecode = bytecodeFile.toString();

const contract = new web3.eth.Contract(abi, contractAddress);

const getTransactionData = async () => {
  await unlockAccount();

  transactionHash = txHash;

  // Get transaction
```

```
  var tx = await web3.eth.getTransaction(transactionHash);

  // Decrypt the transaction input field using the private keys of our node
  var decryptedInput = await web3.eth.getQuorumPayload(tx.input);

  // Decode the hexadecimal representation into the json data
  const decodedInput = await abiDecoder.decodeMethod(decryptedInput);

  console.log("Leyendo Tx:", transactionHash);

  console.log("contenido:", decodedInput.params[1].value);

  fs.writeFile('./config-file.txt', decodedInput.params[1].value, (err) => {
    if (err) {
      console.error('Error al guardar el archivo localmente', err);
      return;
    }
    console.log('archivo de configuración guardado en', 'config-file.txt');
  });
}

async function unlockAccount(){
    const password = '';
    const accounts = await web3.eth.getAccounts();
    const account = await web3.eth.personal.unlockAccount(accounts[0], password, null);
    console.log("Usando cuenta:", accounts[0]);
}

async function main() {
  await getTransactionData(txHash);
}

main();
```