Interoperabilidad de la Historia Clínica Digital por Medio de Blockchain

Carrera de Maestria en Seguridad Informática
Trabajo Final

Autor: **Ing. Pablo E. Bullian**

Director de Trabajo Final: **Juan Pedro Hecht, Ph.D.**

Buenos Aires Cohorte 2020

Universidad de Buenos Aires Facultades de Ciencias Económicas, Ciencias Exactas y Naturales e Ingeniería

Por medio de la presente, el autor manifiesta conocer y aceptar el Reglamento de Tesis vigente y que se hace responsable que la totalidad de los contenidos del presente documento son originales y de su creación exclusiva, o bien pertenecen a terceros u otras fuentes, que han sido adecuadamente referenciados y cuya inclusión no infringe la legislación Nacional e Internacional de Propiedad Intelectual.

FIRMADO Pablo Esteban Bullian DNI 33111349

Interoperabilidad de la Historia Clínica Digital por Medio de Blockchain

Ing. Pablo E. Bullian

1. Resumen

El siguiente trabajo explora una solución posible a la problemática de la interoperabilidad nacional en la salud. Siguiendo la ley promulgada sobre la RED NACIONAL DE INTEROPERABILIDAD EN SALUD[1] se propone y se realizarán pruebas de concepto del uso de Blockchains (Distributed Ledger Technology) en combinación con técnicas de encriptación del estado del arte para brindar una solución que contemple la disponibilidad, integridad y confidencialidad de la información tratada.

1.1. Palabras Clave

Interoperabilidad en la salud, ciberseguridad, Hyperledger, Blockchain, smart contract, IPFS, almacenamiento distribuido

Índice

2. Introducción

2.1. Blockchain

En términos generales blockchain es una cadena inmutable de transacciones mantenida dentro de una red distribuida de nodos. Cada uno de estos nodos mantiene una copia de la cadena aplicando transacciones que han sido validadas por un protocolo de consenso, agrupadas en bloques que incluyen un hash que vincula cada bloque al bloque anterior.

Las aplicaciones más conocidas del blockchain incluyen al Bitcoin[2], una criptomoneda muy famosa y de las más habituales a escuchar en ámbitos de divulgación. También tenemos al Ethereum, una criptomoneda que incluyó el concepto de "smart contracts" creando así una plataforma de aplicaciones distribuidas.

Ambos ejemplos mencionados se pueden clasificar en los denominadas tecnologías de blockchain "public permissionless", que seria redes de público acceso abiertas a cualquier participante para que actúe de forma anónima.

La tecnología de blockchain es muy interesante por muchas de sus características, pero para ampliar su uso debemos contar con algunas de las siguientes características que las redes antes mencionadas no poseen:

- Los participantes deben ser identificables
- Las redes deben tener permiso

- Alto rendimiento entrante de transacciones
- Baja latencia de la confirmación de la transacción
- Privacidad y confidencialidad de transacciones y datos relacionados con transacciones comerciales.

Es aquí donde debemos hablar de otras tecnologías de blockchain para soportar estos requerimientos.

2.2. Hyperledger Fabric

Hyperledger Fabric[3] es una plataforma de tecnología de ledger distribuido (DLT) de código abierto, que ofrece algunas capacidades de diferenciación clave sobre otras plataformas populares de contabilidad distribuida o blockchain.

Un punto clave de diferenciación es que Hyperledger se estableció bajo la Fundación Linux, que a su vez tiene una historia larga y muy exitosa de nutrir proyectos de código abierto bajo una gobernanza abierta que hacen crecer comunidades sólidas y sostenibles y ecosistemas prósperos. Hyperledger está gobernado por un comité directivo técnico diverso, y el proyecto Hyperledger Fabric por un conjunto diverso de mantenedores de múltiples organizaciones. Tiene una comunidad de desarrollo que ha crecido a más de 35 organizaciones y casi 200 desarrolladores desde sus inicios.

Fabric tiene una arquitectura altamente modular y configurable, que permite la innovación, versatilidad y optimización para una amplia gama de casos de uso

de la industria que incluyen banca, finanzas, seguros, atención médica, recursos humanos, cadena de suministro e incluso entrega de música digital.

Fabric es la primera plataforma de blockchain distribuido que admite contratos inteligentes creados en lenguajes de programación de uso general como Java, Go y Node.js, en lugar de lenguajes específicos de dominio restringidos (DSL) como es el caso de Ethereum. Esto significa que la mayoría de las empresas ya tienen las habilidades necesarias para desarrollar contratos inteligentes y no se necesita capacitación adicional para aprender un nuevo idioma o DSL.

La plataforma Fabric también tiene permisos, lo que significa que, a diferencia de una red pública sin permisos, los participantes se conocen entre sí, en lugar de ser anónimos. Esto significa que, si bien los participantes pueden no confiar plenamente entre sí (pueden, por ejemplo, ser competidores en la misma industria), una red puede operarse bajo un modelo de gobernanza que se basa en la confianza que existe entre los participantes, como un acuerdo legal o marco para manejar disputas.

Uno de los diferenciadores más importantes de la plataforma es su soporte para protocolos de consenso conectables que permiten que la plataforma se personalice de manera más efectiva para adaptarse a casos de uso particulares y modelos de confianza. Por ejemplo, cuando se implementa dentro de una sola empresa, o es operado por una autoridad confiable, el consenso totalmente bizantino[4] tolerante a fallas puede considerarse innecesario y un lastre excesivo para el rendimiento y el throughput. En situaciones como esa, un protocolo

de consenso tolerante a fallas(Control Tolerante a Fallas) podría ser más que adecuado, mientras que, en un caso de uso descentralizado de múltiples partes, podría ser necesario un protocolo de consenso tolerante a fallas bizantino (BFT) más tradicional.

Fabric puede aprovechar los protocolos de consenso que no requieren una criptomoneda nativa para incentivar la minería costosa o para impulsar la ejecución inteligente de contratos. Evitar una criptomoneda reduce algunos vectores de riesgo / ataque significativos, y la ausencia de operaciones de minería criptográfica significa que la plataforma se puede implementar con aproximadamente el mismo costo operativo que cualquier otro sistema distribuido.

2.2.1. Modularidad

Hyperledger Fabric[5] ha sido diseñado específicamente para tener una arquitectura modular. Ya sea por consenso conectable, protocolos de administración de identidad conectables como LDAP u OpenID Connect, protocolos de administración de claves o bibliotecas criptográficas, la plataforma se ha diseñado en su núcleo para ser configurada para cumplir con la diversidad de requisitos de casos de uso.

A un alto nivel, Fabric se compone de los siguientes componentes modulares:

- Un servicio de órdenes (ordering service) conectable establece un consenso sobre el orden de las transacciones y luego transmite los bloques a los pares.
- Un proveedor de servicios de membresía conectable es responsable de aso-

ciar entidades en la red con identidades criptográficas.

- Un servicio opcional de "peer-to-peer gossip" difunde la salida de los bloques del servidor de pedidos a otros compañeros.
- Los contratos inteligentes (ȩhaincode"dentro de Hyperledger Fabric) se ejecutan dentro de un entorno de contenedor (por ejemplo, Docker) para el aislamiento. Se pueden escribir en lenguajes de programación estándar, pero no tienen acceso directo al blockchain.
- Una aplicación de políticas de validación y consenso conectable que se puede configurar de forma independiente por aplicación.

2.2.2. Permissioned vs Permissionless Blockchains

En una cadena de bloques permissionless (sin permiso), prácticamente cualquier persona puede participar, y cada participante es anónimo. En tal contexto, no puede haber otra confianza que el estado de la cadena de bloques,luego de cierta "profundidad" (cantidad de bloques posteriores agregados a la cadena), es inmutable. Para mitigar esta falta de confianza, las blockchain sin permiso suelen emplear una criptomoneda nativa "minada.º tarifas de transacción para proporcionar un incentivo económico para compensar los costos extraordinarios de participar en una forma de consenso bizantino tolerante a fallas basado en la "prueba de trabajo"(PoW)[6].

Los blockchain permissioned (autorizados), por otro lado, operan una cadena de bloques entre un conjunto de participantes conocidos, identificados y a menudo auditados que operan bajo un modelo de gobierno que genera un cierto grado

de confianza. Un blockchain permissioned proporciona una forma de asegurar las interacciones entre un grupo de entidades que tienen un objetivo común pero que pueden no confiar plenamente entre sí. Al confiar en las identidades de los participantes, una cadena de bloques autorizada puede utilizar protocolos de consenso más tradicionales tolerantes a fallas (CFT) o tolerantes a fallas bizantinas (BFT) que no requieren una minería costosa.

Además, en un contexto autorizado, se reduce el riesgo de que un participante introduzca intencionalmente código malicioso a través de un contrato inteligente. Primero, los participantes se conocen entre sí y todas las acciones, ya sea enviar transacciones de aplicaciones, modificar la configuración de la red o implementar un contrato inteligente, se registran en la cadena de bloques siguiendo una política de endoso que se estableció para la red y el tipo de transacción relevante. En lugar de ser completamente anónimo, la parte culpable puede identificarse fácilmente y el incidente manejarse de acuerdo con los términos del modelo de gobierno.

2.2.3. Smart Contracts

Un smart contract[7] (contrato inteligente), o lo que Fabric llama "chaincode" (código de cadena), funciona como una aplicación distribuida confiable que obtiene su seguridad / confianza de la cadena de bloques y el consenso subyacente entre los pares.

Hay tres puntos clave que se aplican a los contratos inteligentes, especialmente cuando se aplican a una plataforma:

- Muchos contratos inteligentes se ejecutan simultáneamente en la red,

- Pueden desplegarse dinámicamente (en muchos casos por cualquier persona)
- El código de la aplicación debe tratarse como no confiable, incluso potencialmente malicioso.

La mayoría de las plataformas de blockchain con capacidad para smart contract siguen una arquitectura de ejecución de órdenes en la que el protocolo de consenso:

- válida y ordena las transacciones y luego las propaga a todos los nodos pares,
- cada par luego ejecuta las transacciones secuencialmente.

La arquitectura de order-execute (ordenar-ejecutar) se puede encontrar en prácticamente todos los sistemas blockchain existentes, que van desde plataformas públicas / sin permiso como Ethereum (con consenso basado en PoW) hasta plataformas autorizadas como Tendermint, Chain y Quorum.

Los smart contracts que se ejecutan en una blockchain que opera con la arquitectura de order-execute deben ser deterministas; de lo contrario, es posible que nunca se alcance el consenso. Para abordar el problema del no determinismo, muchas plataformas requieren que los smart contracts se escriban en un lenguaje no estándar o específico del dominio (como Solidity) para que las operaciones no deterministas puedan eliminarse.

Además, dado que todas las transacciones se ejecutan secuencialmente por todos los nodos, el rendimiento y la escala son limitados. El hecho de que el códi-

go de smart contract se ejecute en cada nodo del sistema exige que se tomen medidas complejas para proteger el sistema en general de contratos potencialmente maliciosos a fin de garantizar la resiliencia del sistema en general.

Fabric presenta una nueva arquitectura para transacciones que llamamos execute-order-validate (ejecutar-ordenar-validar). Aborda los desafíos de resiliencia, flexibilidad, escalabilidad, rendimiento y confidencialidad que enfrenta el modelo de ejecución de órdenes al separar el flujo de transacciones en tres pasos:

- ejecutar una transacción y verificar su corrección, aceptandola así
- ordenar transacciones a través de un protocolo de consenso (conectable)
- validar transacciones contra una política de endoso específica de la aplicación antes de comprometerlas en la cadena (ledger)

Este diseño se aparta radicalmente del paradigma de order-execute en el que Fabric ejecuta transacciones antes de llegar a un acuerdo final sobre su orden.

En Fabric, una política de endoso específica de la aplicación especifica qué nodos pares, o cuántos de ellos, deben responder por la ejecución correcta de un contrato inteligente determinado. Por lo tanto, cada transacción solo necesita ser ejecutada (endosada) por el subconjunto de nodos pares necesarios para satisfacer la política de aprobación de la transacción. Esto permite la ejecución en paralelo aumentando el rendimiento general y la escala del sistema. Esta primera fase también elimina cualquier no determinismo, ya que los resultados inconsistentes se pueden filtrar antes de realizar el pedido.

Debido a que hemos eliminado el no determinismo, Fabric es la primera tecnología blockchain que permite el uso de lenguajes de programación estándar.

2.2.4. Privacidad y Confidencialidad

En una red blockchain pública permissionless que aprovecha PoW para su modelo de consenso, las transacciones se ejecutan en cada nodo. Esto significa que no puede haber confidencialidad de los contratos en sí ni de los datos de las transacciones que procesan. Cada transacción, y el código que la implementa, es visible para todos los nodos de la red. En este caso, se cambia la confidencialidad del contrato y los datos por un consenso bizantino tolerante a fallas entregado por PoW.

Esta falta de confidencialidad puede ser problemática para muchos casos de uso. Por ejemplo, en una red de socios de la cadena de suministro, a algunos consumidores se les pueden dar tarifas preferenciales como un medio para solidificar una relación o promover ventas adicionales. Si todos los participantes pueden ver todos los contratos y transacciones, será imposible mantener tales relaciones comerciales en una red completamente transparente.

Para abordar la falta de privacidad y confidencialidad con el fin de cumplir con los requisitos de casos de uso privados, las plataformas blockchain han adoptado una variedad de enfoques.

La encriptación de datos es un método para brindar confidencialidad; sin embargo, en una red sin permisos que aprovecha PoW para su consenso, los datos

cifrados se encuentran en cada nodo. Para muchos casos de uso empresarial, el riesgo de que su información se vea comprometida es inaceptable por mas que aquellos datos se guarden cifrados.
Las pruebas de conocimiento cero (ZKP) son otra área de investigación que se está explorando para abordar este problema, y la compensación aquí es que, en la actualidad, calcular un ZKP requiere un tiempo y recursos computacionales considerables. Por lo tanto, la compensación en este caso es el desempeño por la confidencialidad. En un contexto autorizado que puede aprovechar formas alternativas de consenso, se podrían explorar enfoques que restrinjan la distribución de información confidencial exclusivamente a los nodos autorizados.

Hyperledger Fabric, al ser una plataforma con autorización, permite la confidencialidad a través de su arquitectura de canales y función de datos privados. En los canales, los participantes de una red de Fabric establecen una subred donde cada miembro tiene visibilidad de un conjunto particular de transacciones. Así, sólo aquellos nodos que participan en un canal tienen acceso al contrato inteligente (chaincode) y a los datos transaccionados, preservando la privacidad y confidencialidad de ambos. Los datos privados permiten recopilaciones entre miembros en un canal, lo que permite gran parte de la misma protección que los canales sin la sobrecarga de mantenimiento de crear y mantener un canal separado.

2.2.5. Consenso

El orden de las transacciones se delega a un componente modular de consenso que está lógicamente desacoplado de los pares que ejecutan las transacciones y

mantienen el ledger. En concreto, el "ordering service" (servicio de órdenes) . Dado que el consenso es modular, su implementación se puede adaptar al supuesto de confianza de una implementación o solución en particular. Esta arquitectura modular permite que la plataforma se base en conjuntos de herramientas bien establecidos para pedidos CFT (tolerante a fallas) o BFT (tolerante a fallas bizantino).

Fabric ofrece actualmente una implementación de servicio de órdenes CFT basada en la biblioteca etcd del protocolo Raft[8].

Hay que tener en cuenta también que estos no son mutuamente excluyentes. Una red Fabric puede tener varios servicios de órdenes que admiten diferentes aplicaciones o requisitos de aplicación.

2.2.6. Ordering (ordenes)

Muchas blockchain distribuidas, como Ethereum y Bitcoin, no tienen un sistema de permisos, lo que significa que cualquier nodo puede participar en el proceso de consenso, en el que las transacciones se ordenan y agrupan en bloques. Debido a este hecho, estos sistemas se basan en algoritmos de consenso probabilístico que eventualmente garantizan la consistencia del ledger con un alto grado de probabilidad, pero que aún son vulnerables a ledger divergentes (también conocidos como "bifurcación" del ledger), donde diferentes participantes de la red tienen una visión diferente del orden aceptado de las transacciones.

Hyperledger Fabric funciona de manera diferente. Cuenta con un nodo llamado ordenador (también conocido como "nodo de pedido") que realiza este

pedido de transacciones, que junto con otros nodos de órdenes forma un servicio de órdenes. Debido a que el diseño de Fabric se basa en algoritmos de consenso deterministas, se garantiza que cualquier bloque validado por el par será definitivo y correcto. Los ledger no pueden bifurcar como lo hacen en muchas otras redes de blockchain distribuidas y sin permisos.

Además de promover la aceptación final, separar el respaldo de la ejecución del chaincode (que ocurre en los pares) de la ordenación le da a Fabric ventajas en el rendimiento y la escalabilidad, eliminando los cuellos de botella que pueden ocurrir cuando la ejecución y la ordenación se realizan por los mismos nodos.

Además de su función de ordenación, los ordenadores también mantienen la lista de organizaciones a las que se les permite crear canales. Esta lista de organizaciones se conoce como consortium, y la lista en sí se mantiene en la configuración del canal del sistema de órdenes. De forma predeterminada, esta lista, y el canal en el que vive, solo pueden ser editados por el administrador del ordenador. Tenga en cuenta que es posible que un servicio de órdenes tenga varias de estas listas, lo que convierte al consorcio en un vehículo para la tenencia múltiple de Fabric.

Los ordenadores también imponen un control de acceso básico para los canales, restringiendo quién puede leer y escribir datos en ellos y quién puede configurarlos. Recordando que quien está autorizado para modificar un elemento de configuración en un canal está sujeto a las políticas que los administradores relevantes establecieron cuando crearon el consorcio o el canal. Las transaccio-

nes de configuración son procesadas por el ordenador, ya que necesita conocer el conjunto actual de políticas para ejecutar su forma básica de control de acceso. En este caso, el ordenante procesa la actualización de la configuración para asegurarse de que el solicitante tenga los derechos administrativos adecuados. Si es así, el ordenante válida la solicitud de actualización con la configuración existente, genera una nueva transacción de configuración y la empaqueta en un bloque que se transmite a todos los pares del canal. Los pares luego procesan las transacciones de configuración para verificar que las modificaciones aprobadas por el ordenante satisfacen efectivamente las políticas definidas en el canal.

Todo aquello que va a interactuar con la red de blockchain debe obtener su identidad de un administrador de certificados digitales. En general cada Organización contará con un CA (Certificate Authority) del cual se puede establecer un root CA avalado entre todos y CAs intermedios para cada organización.

2.2.7. Flujo de Transacción

2.2.7.1. Fase 1: Proposal En la primera fase, una aplicación cliente envía una propuesta de transacción a un subconjunto de pares que invocarán un smart contract para producir una actualización propuesta del ledger y luego respaldarán o endosaran los resultados. Los pares que respaldan no aplican la actualización propuesta a su copia del libro mayor en este momento. En cambio, los pares que respaldan devuelven una respuesta de proposal (propuesta) a la aplicación cliente. Las propuestas de transacciones respaldadas finalmente se ordenarán en bloques en la fase dos y luego se distribuirán a todos los pares para la validación final y el commit (aplicación) en la fase tres.

En la Figura 1 la aplicación (A1) genera una transacción (T1) propuesta (P), que envía a través del canal (C) a los peers (P1 y P2). P1 ejecuta S1 con T1 - P generando una respuesta (R1) con una validación (E1). Al mismo tiempo P2 ejecuta S1 y genera R2 con validación E2. Por último A1 recibe T1 con las validaciones E1 y E2.

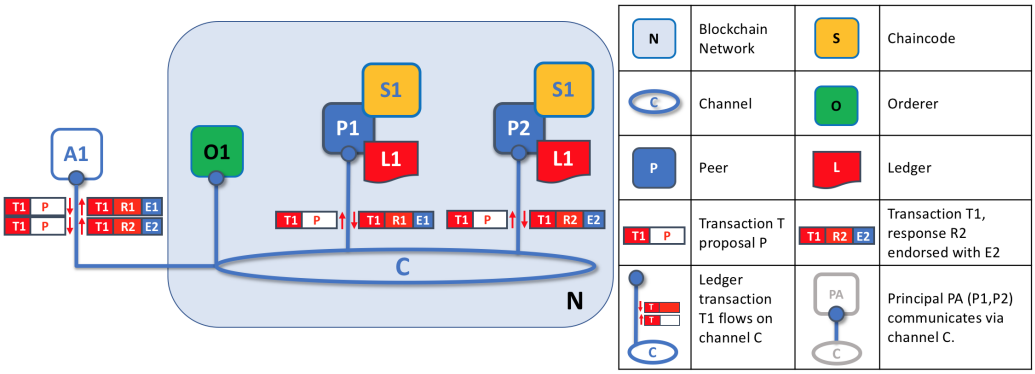

Figura 1: Fase 1 de la Transacción

Cuando una aplicación genera una transacción debe decidir a qué Peers o Pares envía dicha transacción. Esto depende del "endorsement policy" que se define en cada smart contract o chaincode, donde dictamine qué organización o quienes son necesarias para validar la transacción antes de ser aceptada por la red.

Un par valida una respuesta a la propuesta agregando su firma digital y firmando toda la transacción con su clave privada. Esta validación se puede utilizar posteriormente para demostrar que el par de esta organización generó una respuesta particular.

La fase 1 finaliza cuando la aplicación recibe respuestas de propuesta firma-

das de suficientes pares. Las respuestas de la transacción deben reunirse para compararlas antes de que se pueda detectar el no determinismo.

Al final de la fase 1, la aplicación es libre de descartar respuestas de transacciones inconsistentes si así lo desea, terminando efectivamente el flujo de trabajo de la transacción antes de tiempo.

2.2.7.2. Fase 2: Servicio de Órdenes y empaquetamiento de transacciones en bloques Después de la finalización de la primera fase de una transacción, una aplicación cliente ha recibido una respuesta de propuesta de transacción respaldada (o validada) por un conjunto de pares.

En esta fase, los clientes de la aplicación envían transacciones que contienen respuestas de propuestas de transacciones respaldadas a un nodo de servicio de órdenes. El servicio de órdenes crea bloques de transacciones que, en última instancia, se distribuirán a todos los pares del canal para la validación final y el compromiso en la fase tres.

Los nodos de servicio de órdenes reciben transacciones de muchos clientes de aplicaciones diferentes al mismo tiempo. Estos nodos de servicio de órdenes trabajan juntos para formar colectivamente el servicio de órdenes. Su trabajo consiste en organizar lotes de transacciones enviadas en una secuencia bien definida y empaquetarlos en bloques.

El número de transacciones en un bloque depende de los parámetros de

configuración del canal relacionados con el tamaño deseado y la duración máxima transcurrida para un bloque (parámetros BatchSize y BatchTimeout). Luego, los bloques se guardan en el ledger del ordenador y se distribuyen a todos los pares que se han unido al canal. Si un par está inactivo en este momento o se une al canal más tarde, recibirá los bloques después de volver a conectarse a un nodo de servicio de órdenes o al realizar gossip con otro par.

Para comprenderlo con la Figura 2, la aplicación A1 envía la transacción T1 con las validaciones E1 y E2 al ordenador O1. El mismo empaqueta la transacción T1 con otras recibidas por A2, como la Transacción T2 dentro de un mismo bloque B2.

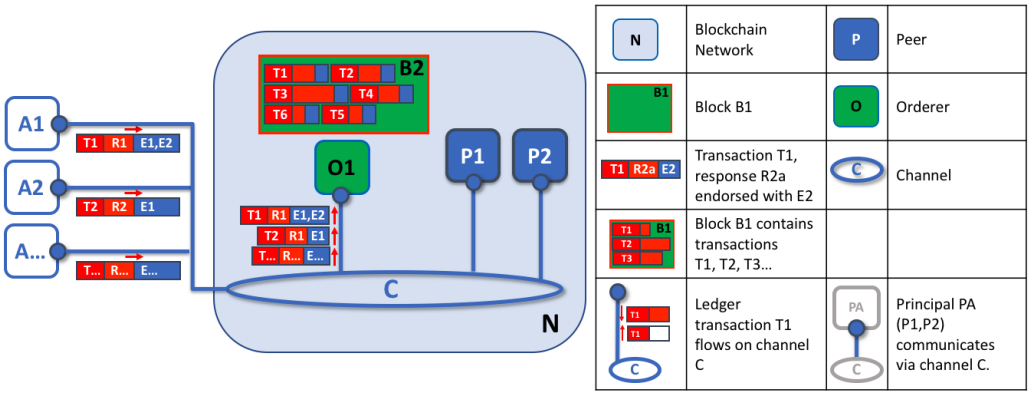

Figura 2: Fase 2 de la Transacción

Vale la pena señalar que la secuencia de transacciones en un bloque no es necesariamente la misma que la orden recibida por el servicio de órdenes, ya que puede haber varios nodos de servicio de órdenes que reciben transacciones aproximadamente al mismo tiempo. Lo importante es que el servicio de órdenes coloca las transacciones en un orden estricto y los pares usarán este orden al validar y confirmar transacciones.

Este orden estricto de las transacciones dentro de los bloques hace que Hyperledger Fabric sea un poco diferente de otras cadenas de bloques donde la misma transacción se puede empaquetar en múltiples bloques diferentes que compiten para formar una cadena. En Hyperledger Fabric, los bloques generados por el servicio de órdenes son definitivos. Una vez que se ha escrito una transacción en un bloque, su posición en el ledger está asegurada de forma inmutable. Como dijimos anteriormente, la finalidad de Hyperledger Fabric significa que no hay bifurcaciones del libro mayor: las transacciones validadas nunca se revertirán ni se eliminarán.

También podemos ver que, mientras que los pares ejecutan contratos inteligentes y procesan transacciones, los ordenadores definitivamente no lo hacen. Cada transacción autorizada que llega a un ordenante se empaqueta mecánicamente en un bloque; el ordenante no juzga el contenido de una transacción (excepto para las transacciones de configuración de canal).

Al final de la fase dos, vemos que los ordenadores han sido responsables de los procesos simples pero vitales de recopilar las actualizaciones de transacciones propuestas, ordenarlas y empaquetarlas en bloques, listas para su distribución.

2.2.7.3. Fase 3: Validación y Aplicación final La tercera fase del flujo de trabajo de la transacción implica la distribución y posterior validación de los bloques del ordenador a los pares, donde se pueden comprometer con el ledger.

La fase 3 comienza con el ordenante distribuyendo bloques a todos los pares conectados a él. También vale la pena señalar que no todos los pares necesitan estar conectados a un ordenador; los pares pueden conectar bloques en cascada a otros pares utilizando el protocolo de gossip.

Cada par validará los bloques distribuidos de forma independiente, pero de forma determinista, asegurando que los ledger permanezcan coherentes. Específicamente, cada peer en el canal validará cada transacción en el bloque para asegurarse de que haya sido respaldada por los peers de la organización requerida, que sus respaldos coincidan y que no haya sido invalidada por otras transacciones comprometidas recientemente que pudo haber estado en vuelo cuando la transacción fue aprobada originalmente. Las transacciones invalidadas aún se conservan en el bloque inmutable creado por el ordenante, pero el par las marca como no válidas y no actualizan el estado del ledger.

En la Figura 3, el ordenador O1 distribuye el bloque B2 a los pares P1 y P2. El P1 procesa el bloque B2 y lo agrega a su ledger (L1). En paralelo P2 válida B2 y también lo agrega sobre su ledger L1. Una vez terminado ambos pares tienen la misma información e informan a las aplicaciones intervinientes la finalización.

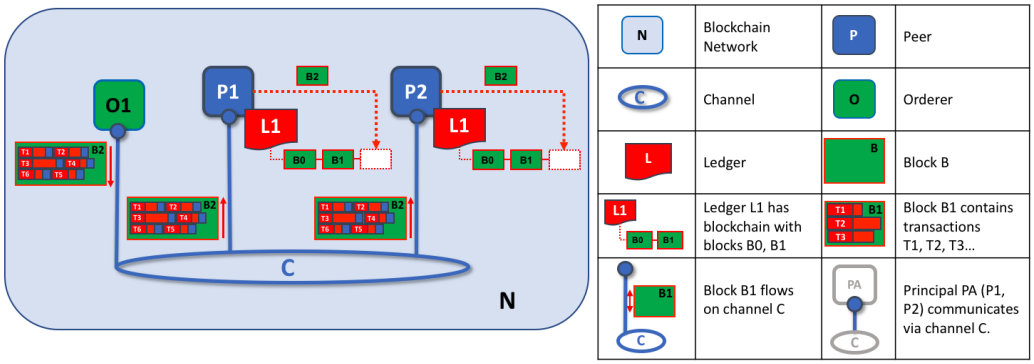

Figura 3: Fase 3 de la Transacción

2.3. InterPlanetary File System (IPFS)

IPFS[9] es un sistema distribuido para almacenar y acceder a archivos, sitios web, aplicaciones y datos.

Es un sistema de archivos distribuido peer-to-peer que busca conectar todos los dispositivos informáticos con el mismo sistema de archivos. De alguna manera, IPFS es similar a la Web, pero IPFS podría verse como una sola subred de BitTorrent, intercambiando objetos dentro de un repositorio GIT. En otras palabras, IPFS proporciona un modelo de almacenamiento de bloques con contenido direccionado de alto rendimiento, con hipervínculos dirigidos al contenido. Esto forma un Merkle generalizado DAG, una estructura de datos sobre la que se pueden construir versiones sistemas de archivos, blockchains e incluso una Web permanente.

IPFS combina una tabla hash distribuida, un intercambio de bloques incentivado y un espacio de nombres autocertificado. IPFS no tiene un solo punto de

falla, y los nodos no necesitan confiar entre sí.

2.3.1. Direccionamiento de contenido

IPFS utiliza un método de direccionamiento de contenido diferente para identificar el contenido. Este enfoque es diferente del que existe en la web hoy en día, donde un servidor que contiene el objeto dirige el contenido.

La dirección de contenido IPFS se resuelve en un objeto IPFS que contiene una lista de enlaces IPFS y algunos datos de contenido. Al agregar archivos enormes a IPFS, se divide en muchos fragmentos más pequeños y la dirección se resuelve en una lista de enlaces IPFS que apuntan a los fragmentos. Este enfoque de direccionamiento de contenido confirma que la dirección siempre devolverá el mismo archivo. Otro beneficio de este enfoque es que siempre que uno de los nodos tenga el contenido, siempre se puede acceder a él desde la red IPFS. Esto resuelve el problema de los enlaces muertos que existe en la Internet de hoy. Los archivos duplicados no ocupan varios espacios porque siempre apuntarán al mismo hash de contenido. Tener solo una copia en la red es suficiente para que se pueda recuperar de la red.

2.3.2. Direcciones

Dada la dirección de contenido, la red IPFS responde con los pares que tienen los objetos requeridos. El objeto y sus enlaces se pueden recibir simultáneamente de varios pares.

Nuestros pares pueden compartir el objeto después de que IPFS lo devuelva

de la red. El nodo también puede verificar que el contenido no haya sido manipulado porque el valor hash se asigna al contenido en la red IPFS. Dado que el contenido del archivo genera la dirección de contenido (hash), cambia cada vez que se modifica un archivo.

No es conveniente compartir una nueva dirección de contenido cada vez que cambia el archivo. Para abordar este problema, IPFS tiene un concepto de IPNS, o el Sistema de Nombres Interplanetarios.

2.3.3. CIDs

Un identificador de contenido, o CID, es una etiqueta que se usa para señalar material en IPFS. No indica dónde se almacena el contenido, pero forma una especie de dirección basada en el contenido en sí. Los CID son cortos, independientemente del tamaño de su contenido subyacente.

Los CID se basan en el hash criptográfico del contenido. Eso significa:

- Cualquier diferencia en el contenido producirá un CID diferente y
- El mismo contenido agregado a dos nodos IPFS diferentes usando la misma configuración producirá el mismo CID.

IPFS utiliza el algoritmo de hashing sha-256 de forma predeterminada, pero hay soporte para muchos otros algoritmos.

Los hashes criptográficos tienen características muy importantes:

- determinista: el mismo mensaje de entrada siempre devuelve exactamente el mismo hash de salida

- no correlacionado: un pequeño cambio en el mensaje debería generar un hash completamente diferente
- único: no es factible generar el mismo hash a partir de dos mensajes diferentes
- unidireccional: no es factible adivinar o calcular el mensaje de entrada a partir de su hash

Resulta que estas características también significan que podemos usar un hash criptográfico para identificar cualquier dato: el hash es único para los datos a partir de los cuales lo calculamos, y no es demasiado largo (un hash tiene una longitud fija, por lo que el SHA-256 hash de un archivo de video de un gigabyte sigue siendo de solo 32 bytes), por lo que enviarlo por la red no consume muchos recursos.

Eso es fundamental para un sistema distribuido como IPFS, donde queremos poder almacenar y recuperar datos de muchos lugares. Una computadora que ejecuta IPFS puede preguntar a todos los pares a los que está conectada si tienen un archivo con un hash en particular y, si uno de ellos lo tiene, devuelve el archivo completo. Sin un identificador único y corto como un hash criptográfico, eso no sería posible. Esta técnica se denomina direccionamiento de contenido, porque el contenido en sí se usa para formar una dirección, en lugar de información sobre la computadora y la ubicación del disco en la que se almacena.

2.3.4. Persistencia y Permanencia

Los nodos de la red IPFS pueden almacenar en caché automáticamente los recursos que descargan y mantener esos recursos disponibles para otros nodos. Este sistema depende de que los nodos estén dispuestos y sean capaces de almacenar en caché y compartir recursos con la red. El almacenamiento es finito, por lo que los nodos deben borrar algunos de sus recursos previamente almacenados en caché para dejar espacio para nuevos recursos. Este proceso se llama recolección de basura.

Para garantizar que los datos persistan en IPFS y no se eliminen durante la recolección de basura, los datos se pueden anclar a uno o más nodos IPFS. La fijación (pinning) le brinda control sobre el espacio en disco y la retención de datos. Como tal, debe usar ese control para anclar cualquier contenido que desee mantener en IPFS indefinidamente.

2.3.5. Nodos IPFS

Un NodeId identifica un nodo en el sistema IPFS. Un NodeId no es más que el hash de su clave pública. Los nodos pueden cambiar su NodeId en cualquier momento, pero están incentivados a seguir siendo los mismos. Los nodos almacenan objetos (de su interés) en su almacenamiento local. Estos objetos representan archivos y otras estructuras de datos en IPFS. Todos los nodos mantienen un DHT que se usa para encontrar: Dirección de red de otros pares en la red y compañeros que pueden servir a un objeto en particular

Este DHT permite que IPFS encuentre pares que puedan servir un objeto y poder llegar al par a través de la red.

2.3.6. Distributed Hash Tables (DHTs)

Una tabla hash distribuida (DHT) es un sistema distribuido para asignar claves a valores. En IPFS, el DHT se utiliza como componente fundamental del sistema de enrutamiento de contenido y actúa como un cruce entre un catálogo y un sistema de navegación. Asigna lo que el usuario busca al par que almacena el contenido coincidente. Puede entenderse como una tabla enorme que almacena quién tiene qué datos. Hay tres tipos de emparejamientos clave-valor que se asignan mediante DHT:

Tipo	Propósito	**Utilizado por**
Catálogo de provedor	Mapea un identificador de datos (un multihash en este caso) a un peer que ha publicado que el tiene el contenido y esta disponible.	- IPFS para buscar contenido.
IPNS records	Mapea una llave IPNS (i.e., el hash de la clave pública) a un IPNS record (un puntero firmado del estilo /ipfs/bafyxyz...)	- IPNS
Peer records	Mapea un peerID a un set de multiaddresses donde el peer pueda ser accesible	- IPFS cuando conoce el peer pero no su dirección para acceder.

Cuadro 1: Tabla de uso de DHT

2.3.7. Kademlia

El algoritmo Kademlia[10] ha existido por un tiempo, y su propósito es construir un DHT sobre tres parámetros del sistema:

- Un espacio de direcciones como una forma en que todos los pares de la red pueden identificarse de forma única. En IPFS, estos son todos los números del 0 al 2^{256-1}

- Una métrica para ordenar los pares en el espacio de direcciones y, por lo tanto, visualizar todos los pares a lo largo de una línea ordenada de menor a mayor. IPFS toma SHA256 (PeerID) y lo interpreta como un número entero entre 0 y 2^{256-1}

- Una proyección que tomará una clave de registro y calculará una posición en el espacio de direcciones donde el par o pares más idóneos para almacenar el registro deberían estar cerca. IPFS utiliza SHA256 (clave de registro).

Tener este espacio de direcciones y una métrica de ordenación entre pares permite buscar en la red como si fuera una lista ordenada. En particular, podemos convertir el sistema en algo así como una lista de omisión(skip-list) donde un par conoce a otros pares con distancias de alrededor de 1,2,4,8 ... lejos de él. Esto nos permitirá buscar en la lista en un tiempo logarítmico en el tamaño de la red, tiempo de búsqueda $O(log(N))$

A diferencia de una lista de omisión, Kademlia es algo inestable ya que los pares pueden unirse, salir y volver a unirse a la red en cualquier momento. Para lidiar con la naturaleza inestable del sistema, un par de Kademlia no solo mantiene vínculos con los pares a una distancia 1,2,4,8 ... lejos de él. En cambio, por cada múltiplo de 2 de distancia, mantiene hasta K enlaces. En IPFS K=20. Por

ejemplo, en lugar de que un par mantenga un solo enlace 128 alejado, mantendría 20 enlaces que están entre 65 y 128 alejados.

La selección de parámetros de toda la red como K no es arbitraria. Se determina en función de la rotación promedio observada en la red y la frecuencia con la que la red volverá a publicar la información. Los parámetros del sistema, como K, se calculan para maximizar la probabilidad de que la red permanezca conectada y de que no se pierdan datos mientras se mantiene la latencia deseada para las consultas y se asume que las observaciones de abandono promedio permanecen constantes. Estos parámetros del sistema y de la red impulsan las decisiones tomadas en los dos componentes principales de Kademlia: la tabla de enrutamiento, que rastrea todos esos enlaces en la red, y el algoritmo de búsqueda, que determina cómo atravesar esos enlaces para almacenar y recuperar datos.

2.3.8. Enrutamiento

Los nodos IPFS requieren un sistema de enrutamiento[11] que pueda encontrar direcciones de red de otros pares y pares que pueden servir a objetos particulares. IPFS logra esto usando un DSHT basado en S/Kademlia[10] y Coral[12].

El tamaño de los objetos y los patrones de uso de IPFS son similares a Coral[12], por lo que IPFS DHT hace una distinción para los valores almacenados en función de su tamaño. Los valores pequeños (iguales o inferiores a 1 KB) se almacenan directamente en el DHT.

Para valores más grandes, el DHT almacena referencias, que son los NodeIds

de los pares que pueden servir el bloque.

IPFS puede utilizar cualquier red; no se basa ni asume acceso a IP. Esto permite que IPFS se utilice en redes superpuestas (Por ejemplo SCTP).

2.3.9. IPFS Gateways

La implementación de IPFS busca incluir soporte nativo de IPFS en todos los navegadores y herramientas populares. Los gateways (puertas de enlace) proporcionan soluciones para las aplicaciones que aún no son compatibles con IPFS de forma nativa.

Por ejemplo, las implementaciones basadas en HTTP traducen a códigos de acceso del protocolo HTTP el acceso a los datos de la red IPFS. Donde utilizar librerías o implementaciones comúnmente utilizadas en la internet permiten el acceso a esta red.

3. Caso de Uso

3.1. Introducción

El diseño propuesto se basa en la utilización del blockchain privado entre distintos organismos de salud, para que el usuario (paciente) inicie bajo su consentimiento una transferencia del historial médico hacia otra organización donde haya un previo requerimiento de un médico o especialista que lo requiera.

Diversos países cuentan con normativas muy estrictas sobre la confidencialidad de los registros médicos (Por ejemplo HIPAA en Estados Unidos) por lo tanto es importante que dicha información sea visualizada solo por las partes aprobadas por el usuario, que se conserve la confidencialidad y la integridad del mismo tanto durante su transporte y su alojamiento.
A continuación se detallan las fases del diseño propuesto.

3.2. Requisitos Previos

La arquitectura se basa en un Certificate Authority confiable para todas las organizaciones, que generará un certificado root o raíz. A partir de este certificado root, se podrán generar certificados intermedios para cada peer u organización que deseen o estén habilitadas para participar. Cada usuario que ejecute acciones por medio de un peer deberá contar con un certificado público y privado firmado por el mismo root o por los certificados intermedios.

Agregado a esto, como previamente desarrollado, se necesita de los ordering

services donde se desarrolla el consenso sobre el blockchain y la creación de un canal común entre las organizaciones donde se instalarán los smart contracts.

3.3. Fase 1

Los médicos o demás actores que deban interactuar con los registros de salud deberán generar un par de claves pública y privadas siguiendo las recomendaciones de los mejores algoritmos y tamaño de llaves del momento con un vencimiento anual (un par de llaves distinto al generado para participar en el blockchain que no necesita estar firmado por el root CA).

Una vez generados se subirá a el blockchain el smart contract denominado "PKI" un registro con los datos para poder hacer la búsqueda pública (pública para las organizaciones participantes) del actor donde contenga su clave pública.

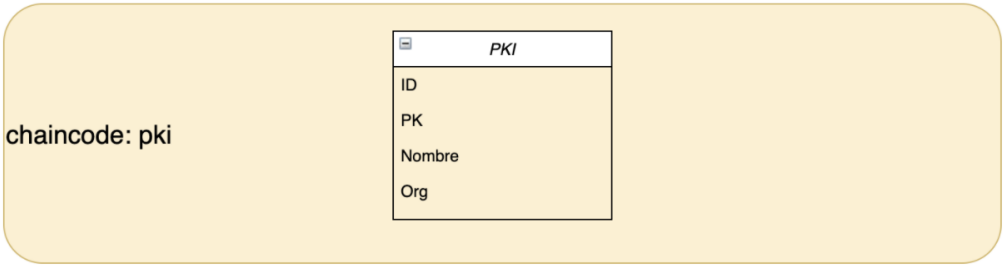

Figura 4: PKI datos

Esta transacción es validada por la organización donde sea miembro. Se recomienda seguir los procedimientos y normativas de la firma digital en Argentina[13].

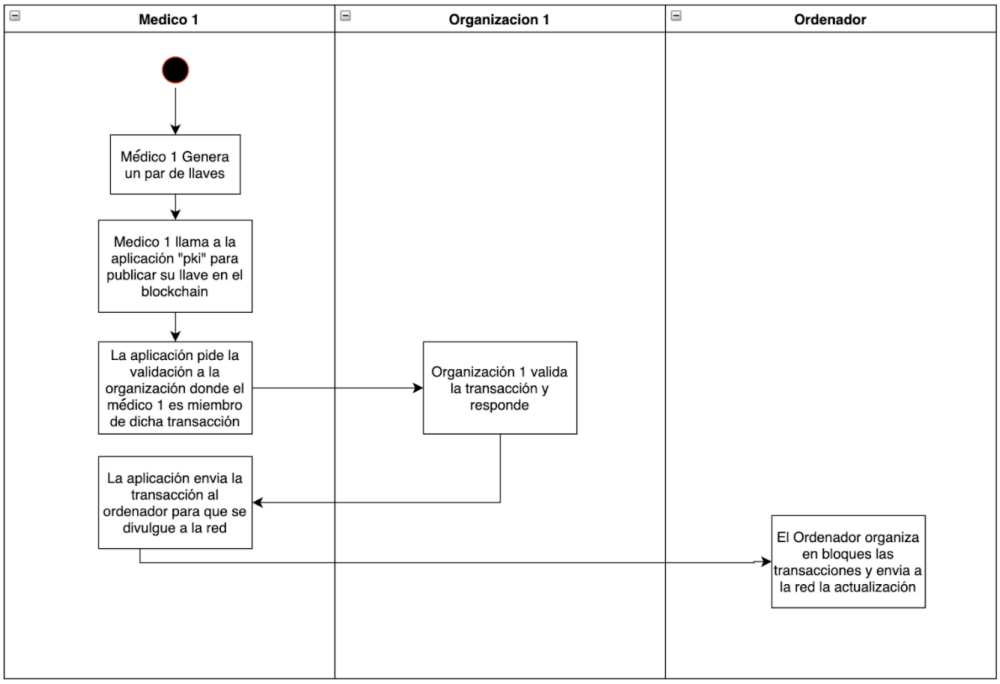

Figura 5: Diagrama de Flujo

3.4. Fase 2

El usuario necesita transferir su historial médico que está resguardado por la organización 1 para ser leído por el médico 2 de la organización 2. El usuario le provee un PIN de seguridad a la organización 1.

Al recibir el pedido, la organización 1, busca sobre el blockchain los smart contract "PKI" del médico 2 para obtener la llave pública.

La organización 1 cifra con la llave pública el historial médico y lo sube a una IPFS donde obtiene un hash del registro que va a ser el identificador para que otros actores puedan encontrarlo.

La organización 1 ejecuta la aplicación del smart contract "Contrato" la cual genera una estructura privada con la información del hash, el PIN, un Salt (el salt es utilizado para que no se pueda adivinar a través del hash el registro, ya que este primero es información pública) y algunos datos relevantes del paciente y el médico 2. Esta misma aplicación genera un registro "público" con el registro de la organización que mantiene ese registro privado, el identificador del médico 2 y un ID único.

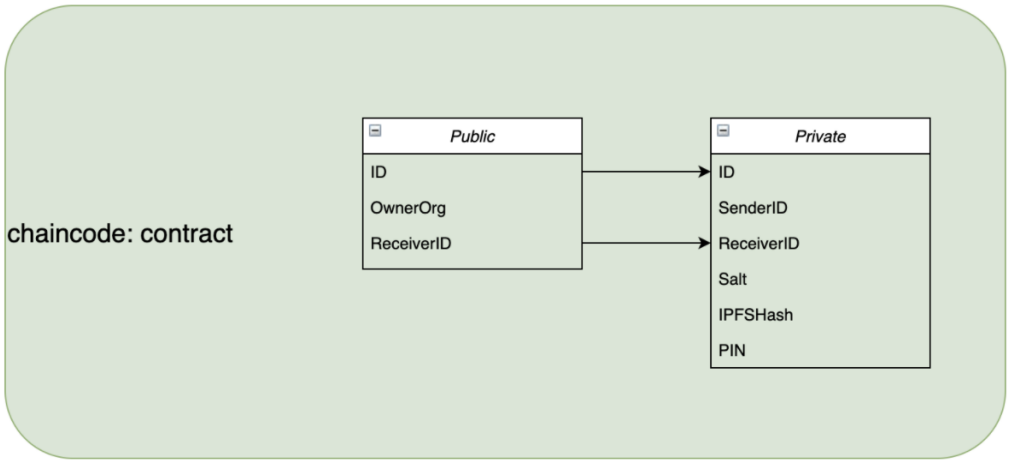

Figura 6: Fase 2 estructura de datos

La organización 1 informa del ID del registro generado al paciente.

3.5. Fase 3

El paciente personalmente entrega la información de toda la estructura de datos a la organización 2, mediante la lectura de un código QR o similar sin

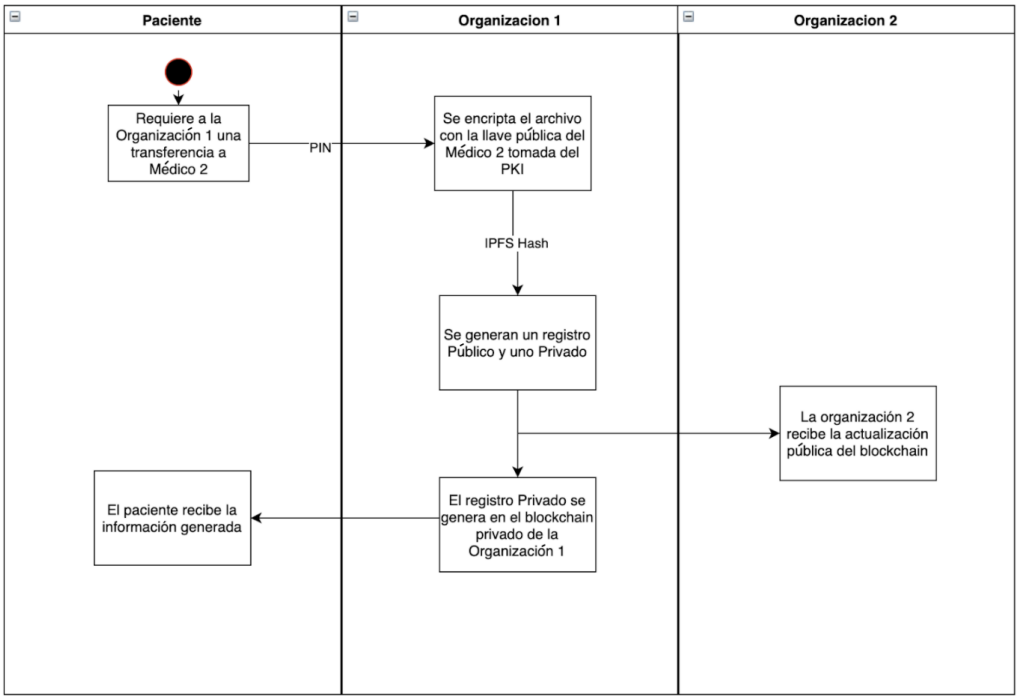

Figura 7: Fase 2 Diagrama de Flujo

revelar el PIN.

El paciente al ser atendido por el médico, le entrega el PIN el cual al ser ingresado en el sistema de la organización 2, se genera un requerimiento de transferencia de la estructura privada del registro que existe en la organización 1 hacia el blockchain privado de la organización 2.

La organización 2 ingresa el mismo PIN dentro del pedido de transacción para que sea validado por los ordenadores del blockchain junto a los demás datos.

El pedido de transacción llega a los ordenadores, donde se validan por el

smart contract comparando el hash de ambas estructuras, si estas coinciden, se transfiere el registro privado de la organización 1 a la organización 2.

En este momento los miembros de la organización 2, cómo el médico 2, pueden ver los datos del registro privado, y así consultar en el IPFS el archivo cifrado.

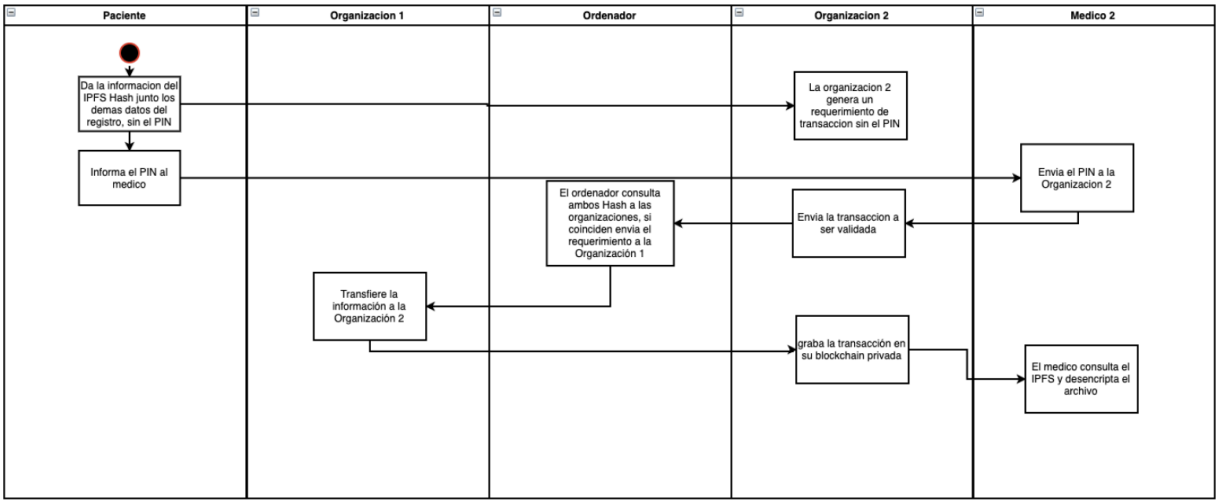

Figura 8: Fase 3 Diagrama de Flujo

Para finalizar, el médico 2 desencripta el archivo cifrado con su llave privada.

3.6. IPFS Clusters

Para la implementación del IPFS se propone utilizar una solución con clusters de nodos IPFS en cada organización para obtener dos puntos muy importantes para el almacenamiento de PHI o datos sensibles que son:

- Redundancia de la información

- Alta disponibilidad.

Si se decide utilizar un solo nodo de IPFS por organización y este no está disponible por alguna falla del datacenter o del software en si, se dañaria la disponibilidad de todo el sistema.

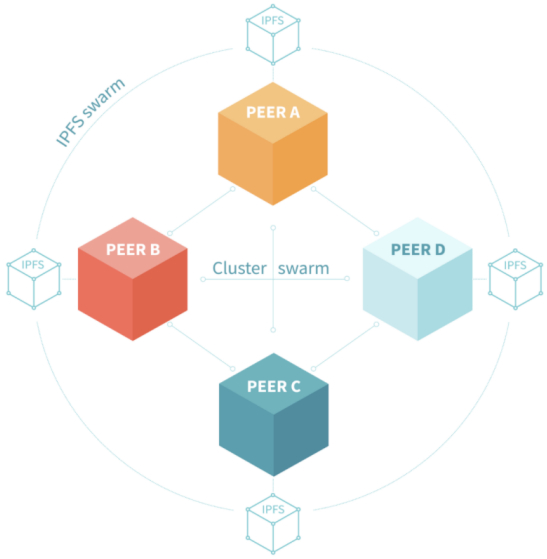

Figura 9: Diagrama de IPFS con peers dentro de un cluster

Cada organización contará con un cluster. Se recomienda que los nodos se encuentren idealmente en distintos datacenters que no compartan una misma región física, conexiones eléctricas o sistemas de conectividad para disminuir el riesgo de que una falla en algunos de estos sistemas esenciales afecte más de un datacenter y deje sin servicio el cluster.

Cada nodo del cluster se conectará como es normal para IPFS con nodos en otros clusters de otras organizaciones pero de forma privada por medio de un secreto compartido. La red de IPFS pública no será utilizada, sólo así su protocolo.

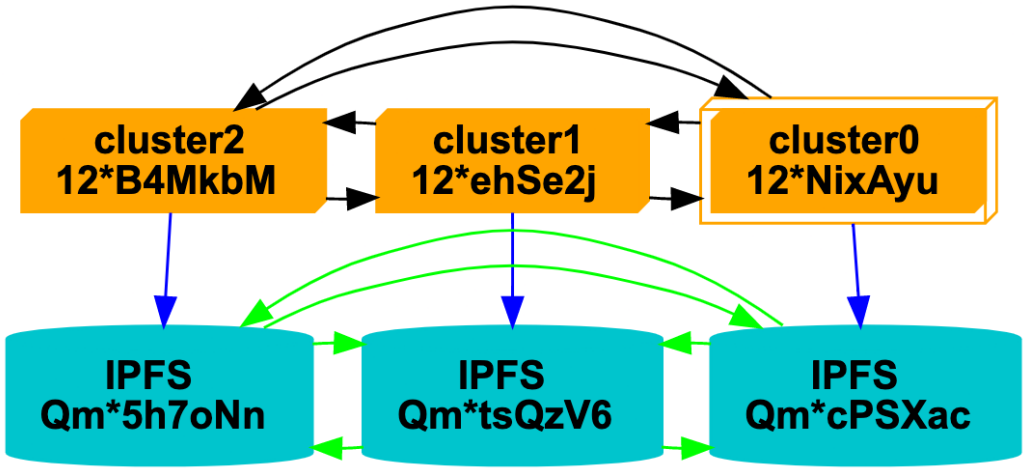

Figura 10: Ejemplo de cluster entre 3 organizaciones con un peer en cada cluster. En verde las conexiones de los peers por IPFS. En azul las conexiones de los clusters con sus peers. En negro las conexiones entre si de los clusters.

4. Conclusión

El trabajo abordó la problemática de una transmisión y cómo compartir datos sensibles como historiales médicos entre distintas organizaciones para responder con la tríada CIA (confidencialidad, Integridad y disponibilidad).

Con el uso de tecnología de blockchain podemos responder a los problemas de integridad y también de trazabilidad de estas transacciones. Además se indaga en tecnologías de blockchain más allá de las típicas monetizadas que utilizan proof-of-work las cuales las hacen poco prácticas y de alto consumo energético y de dispositivos de cálculo como GPUs, dando lugar al uso de tecnologías basadas en el consenso entre pares.

Con la implementación de encriptación en un sistema PKI respaldado por la

misma blockchain logramos mantener la confidencialidad de estos datos y asegurar que solo sean accesibles a quienes cuenten con acceso autorizado previamente por las organizaciones y los mismos pacientes.

Para responder a la disponibilidad se propuso el uso de sistemas distribuidos para compartir datos digitales como es el IPFS. Donde se indaga en la implementación privada pero entre distintas organizaciones previamente autorizadas con el uso de IPFS Cluster.

El trabajo demuestra también en su anexo la prueba de concepto (PoC) de lo propuesto por medio de smartcontracts o chaincode basado en hyperledger, una tecnología de blockchain previamente detallada en el capítulo 1.

5. Anexo

5.1. Chaincode/SmartContract en GO del blockchain PKI

```
package chaincode

import (
  "encoding/json"
  "fmt"

  "github.com/hyperledger/fabric-contract-api-go/contractapi"
)

// funciones para manejar un Asset
type SmartContract struct {
  contractapi.Contract
}

// Descripcion del Asset para el PKI
type Asset struct {
  ID     string `json:"ID"`
  PK     string `json:"pk"`
  Nombre string `json:"nombre"`
  Org    string `json:"org"`
}
```

```
// Inicializacion del ledger con 3 assets (medicos) con sus
    llaves publicas
func (s *SmartContract) InitLedger(ctx
    contractapi.TransactionContextInterface) error {
  assets := []Asset{
    {ID: "asset1", PK:
        "MIGbMBAGByqGSM49AgEGBSuBBAAjA4GGAAQAwy12cB+LhNVTT5If/XBdVlyutUOO6aQvH68cITIB
        Nombre: "Juan Alberto", Org: "UNSAM"},
    {ID: "asset2", PK:
        "MIGbMBAGByqGSM49AgEGBSuBBAAjA4GGAAQBLUOV109tAfZoGAmG4HAgjuN3agiS+9vpn7WOrchQ
        Nombre: "Sergio Martin", Org: "UBA"},
    {ID: "asset3", PK:
        "MIGbMBAGByqGSM49AgEGBSuBBAAjA4GGAAQB+m2EQg9Om8VCchKGRgLeUXP5ypR1Ny6lDf09fbpu
        Nombre: "Facundo Armando", Org: "UNSAM"},
  }

  for _, asset := range assets {
    assetJSON, err := json.Marshal(asset)
    if err != nil {
      return err
    }

    err = ctx.GetStub().PutState(asset.ID, assetJSON)
    if err != nil {
      return fmt.Errorf("Error al crear. %v", err)
    }
```

```
  }

  return nil
}

// Creacion de un nuevo Asset.
func (s *SmartContract) CreateAsset(ctx
    contractapi.TransactionContextInterface, id string, nombre
    string, pk string, org string) error {
  exists, err := s.AssetExists(ctx, id)
  if err != nil {
    return err
  }
  if exists {
    return fmt.Errorf("el asset %s ya existe", id)
  }

  asset := Asset{
    ID:     id,
    PK:     pk,
    Nombre: nombre,
    Org:    org,
  }
  assetJSON, err := json.Marshal(asset)
  if err != nil {
    return err
```

```
  }

  return ctx.GetStub().PutState(id, assetJSON)
}

// ReadAsset devuelve un asset buscado por su ID.
func (s *SmartContract) ReadAsset(ctx
    contractapi.TransactionContextInterface, id string) (*Asset,
    error) {
  assetJSON, err := ctx.GetStub().GetState(id)
  if err != nil {
    return nil, fmt.Errorf("Error de lectura: %v", err)
  }
  if assetJSON == nil {
    return nil, fmt.Errorf("el asset %s no existe", id)
  }

  var asset Asset
  err = json.Unmarshal(assetJSON, &asset)
  if err != nil {
    return nil, err
  }

  return &asset, nil
}
```

```
// Edita un asset que ya existe.
func (s *SmartContract) UpdateAsset(ctx
    contractapi.TransactionContextInterface, id string, nombre
    string, pk string, org string) error {
  exists, err := s.AssetExists(ctx, id)
  if err != nil {
    return err
  }
  if !exists {
    return fmt.Errorf("el asset %s no existe", id)
  }

  // reescritura
  asset := Asset{
    ID:     id,
    PK:     pk,
    Nombre: nombre,
    Org:    org,
  }
  assetJSON, err := json.Marshal(asset)
  if err != nil {
    return err
  }

  return ctx.GetStub().PutState(id, assetJSON)
}
```

```
// Elimina un asset existente del ledger.
func (s *SmartContract) DeleteAsset(ctx
    contractapi.TransactionContextInterface, id string) error {
  exists, err := s.AssetExists(ctx, id)
  if err != nil {
    return err
  }
  if !exists {
    return fmt.Errorf("el asset %s no existe", id)
  }

  return ctx.GetStub().DelState(id)
}

// Chequeo Booleano sobre la existencia de un asset
func (s *SmartContract) AssetExists(ctx
    contractapi.TransactionContextInterface, id string) (bool,
    error) {
  assetJSON, err := ctx.GetStub().GetState(id)
  if err != nil {
    return false, fmt.Errorf("Error de lectura: %v", err)
  }

  return assetJSON != nil, nil
}
```

```
// TransferAsset actualiza la organizacion donde pertenece del
    asset.
func (s *SmartContract) TransferAsset(ctx
    contractapi.TransactionContextInterface, id string, newOwner
    string) error {
  asset, err := s.ReadAsset(ctx, id)
  if err != nil {
    return err
  }

  asset.Org = newOwner
  assetJSON, err := json.Marshal(asset)
  if err != nil {
    return err
  }

  return ctx.GetStub().PutState(id, assetJSON)
}

// GetAllAssets devuelve todos los assets
func (s *SmartContract) GetAllAssets(ctx
    contractapi.TransactionContextInterface) ([]*Asset, error) {

  resultsIterator, err := ctx.GetStub().GetStateByRange("", "")
  if err != nil {
```

```
    return nil, err
  }
  defer resultsIterator.Close()

  var assets []*Asset
  for resultsIterator.HasNext() {
    queryResponse, err := resultsIterator.Next()
    if err != nil {
      return nil, err
    }

    var asset Asset
    err = json.Unmarshal(queryResponse.Value, &asset)
    if err != nil {
      return nil, err
    }
    assets = append(assets, &asset)
  }

  return assets, nil
}
```

5.2. Aplicacion en Javascript para interactuar en blockchain PKI

```
'use strict';

const { Gateway, Wallets } = require('fabric-network');
const FabricCAServices = require('fabric-ca-client');
const path = require('path');
const { buildCAClient, registerAndEnrollUser, enrollAdmin } =
    require('../../test-application/javascript/CAUtil.js');
const { buildCCPOrg1, buildWallet } =
    require('../../test-application/javascript/AppUtil.js');

const channelName = 'mychannel';
const chaincodeName = 'pki';
const mspOrg1 = 'Org1MSP';
const walletPath = path.join(__dirname, 'wallet');
const org1UserId = 'appUser_pki';

function prettyJSONString(inputString) {
  return JSON.stringify(JSON.parse(inputString), null, 2);
}
//comienzo
async function main() {
  try {
    // setea el connection profile
    const ccp = buildCCPOrg1();
```

```
//instancia del CA
const caClient = buildCAClient(FabricCAServices, ccp,
    'ca.org1.example.com');

// creacion de wallet de usuarios
const wallet = await buildWallet(Wallets, walletPath);

// enrolamiento del admin
await enrollAdmin(caClient, wallet, mspOrg1);

// registracion de un usuario de la org1
await registerAndEnrollUser(caClient, wallet, mspOrg1,
    org1UserId, 'org1.department1');

// Creacion del gateway para las organizaciones
const gateway = new Gateway();

try {

  await gateway.connect(ccp, {
    wallet,
    identity: org1UserId,
    discovery: { enabled: true, asLocalhost: true } });

  // creacion de la instancia de red en el gateway creado
```

```
const network = await gateway.getNetwork(channelName);

// obtencion del contrato desde la red
const contract = network.getContract(chaincodeName);

// Se corre la funcion InitLedger del chaincode PKI
console.log('\n--> Transaccion: InitLedger, creacion de
    assets inicial');
await contract.submitTransaction('InitLedger');
console.log('*** Resultado: committed');

// Corre Get all Assets para revisar el resultado anterior
console.log('\n--> Transaccion: GetAllAssets, Trae todos
    los assets');
let result = await
    contract.evaluateTransaction('GetAllAssets');
console.log(`*** Resultado:
    ${prettyJSONString(result.toString())}`);

// Corre AssetExists para revisar el resultado anterior

console.log('\n--> Transaccion: AssetExists, funcion
    devuelve verdadero si existe');
result = await contract.evaluateTransaction('AssetExists',
    'asset1');
```

```
console.log(`*** Resultado:
    ${prettyJSONString(result.toString())}`);


// Corre ReadAsset para revisar el resultado anterior

console.log('\n--> Transaccion: ReadAsset, funcion trae el
    asset1');
result = await contract.evaluateTransaction('ReadAsset',
    'asset1');
console.log(`*** Resultado:
    ${prettyJSONString(result.toString())}`);

try {
//Prueba forzando un error (trata de actualizar un asset no
    existente)
  console.log('\n--> Transaccion: UpdateAsset actualiza el
      asset70 que no existe aun');
  await contract.submitTransaction('UpdateAsset',
      'asset70', 'PK XXXX', 'Juan Perez', 'UNSAM');
  console.log('******** ERROR');
} catch (error) {
  console.log(`*** Error Capturado: \n ${error}`);
}
// TransferAsset cambia la org del medico 1 a UCA desde UBA
```

```
        console.log('\n--> Transaccion: TransferAsset cambia el org
            a UCA');
        await contract.submitTransaction('TransferAsset', 'asset1',
            'UCA');
        console.log('*** Resultado: correcto');

        // Corre ReadAsset para revisar el resultado anterior

        console.log('\n--> Transaccion: ReadAsset');
        result = await contract.evaluateTransaction('ReadAsset',
            'asset1');
        console.log(`*** Resultado:
            ${prettyJSONString(result.toString())}`);
      } finally {
        // deconecta todas las sessiones de la red
        gateway.disconnect();
      }
    } catch (error) {
      console.error(`******** FALLA en correr la app: ${error}`);
    }
}

main();
```

5.3. Aplicacion en Javascript para interactuar en blockchain Contrato

5.3.1. Creación de assets

```
  'use strict';

const { Gateway, Wallets } = require('fabric-network');
const FabricCAServices = require('fabric-ca-client');
const path = require('path');
const { buildCAClient, registerAndEnrollUser, enrollAdmin } =
    require('../../test-application/javascript/CAUtil.js');
const { buildCCPOrg1, buildCCPOrg2, buildWallet } =
    require('../../test-application/javascript/AppUtil.js');

const channelName = 'mychannel';
const chaincodeName = 'contract';

const org1 = 'Org1MSP';
const org2 = 'Org2MSP';
const Org1UserId = 'appUser1';
const Org2UserId = 'appUser2';

const RED = '\x1b[31m\n';
const GREEN = '\x1b[32m\n';
const RESET = '\x1b[0m';
// inicializacion de gateway de las org
```

```
async function initGatewayForOrg1() {
   console.log(`${GREEN}--> Fabric client user & Gateway init:
       Using Org1 identity to Org1 Peer${RESET}`);
   //creacion del connection profile de la org1
   const ccpOrg1 = buildCCPOrg1();

   const caOrg1Client = buildCAClient(FabricCAServices, ccpOrg1,
       'ca.org1.example.com');
   //wallet
   const walletPathOrg1 = path.join(__dirname, 'wallet', 'org1');
   const walletOrg1 = await buildWallet(Wallets, walletPathOrg1);

   // registracion el user principal
   await enrollAdmin(caOrg1Client, walletOrg1, org1);

   await registerAndEnrollUser(caOrg1Client, walletOrg1, org1,
       Org1UserId, 'org1.department1');

   try {

      const gatewayOrg1 = new Gateway();

      await gatewayOrg1.connect(ccpOrg1,
         { wallet: walletOrg1, identity: Org1UserId, discovery: {
             enabled: true, asLocalhost: true } });
```

```
        return gatewayOrg1;
    } catch (error) {
        console.error(`Error en connecion con el gw desde Org1:
            ${error}`);
        process.exit(1);
    }
}

async function initGatewayForOrg2() {
    console.log(`${GREEN}--> Fabric client user & Gateway init:
        Using Org2 identity to Org2 Peer${RESET}`);
    const ccpOrg2 = buildCCPOrg2();
    const caOrg2Client = buildCAClient(FabricCAServices, ccpOrg2,
        'ca.org2.example.com');

    const walletPathOrg2 = path.join(__dirname, 'wallet', 'org2');
    const walletOrg2 = await buildWallet(Wallets, walletPathOrg2);

    await enrollAdmin(caOrg2Client, walletOrg2, org2);
    await registerAndEnrollUser(caOrg2Client, walletOrg2, org2,
        Org2UserId, 'org2.department1');

    try {
        // nuevo Gateway para org 1
        const gatewayOrg2 = new Gateway();
        await gatewayOrg2.connect(ccpOrg2,
```

```
        { wallet: walletOrg2, identity: Org2UserId, discovery: {
            enabled: true, asLocalhost: true } });

      return gatewayOrg2;
   } catch (error) {
      console.error(`Error in connecting to gateway for Org2:
          ${error}`);
      process.exit(1);
   }
}

async function readPrivateAsset(assetKey, org, contract) {
   console.log(`${GREEN}--> Transaccion: GetAssetPrivateProperties,
       - ${assetKey} desde organizacion ${org}${RESET}`);
   try {a
      const resultBuffer = await
          contract.evaluateTransaction('GetAssetPrivateProperties',
          assetKey);
      const asset = JSON.parse(resultBuffer.toString('utf8'));
      console.log(`*** Result: GetAssetPrivateProperties,
          ${JSON.stringify(asset)}`);

   } catch (evalError) {
      console.log(`*** Falla: error evaluateTransaction
          readPrivateAsset: ${evalError}`);
   }
```

```
}

async function readBidPrice(assetKey, org, contract) {
   console.log(`${GREEN}--> Transaccion: GetAssetBidPrice, -
       ${assetKey} desde organizacion ${org}${RESET}`);
   try {
      const resultBuffer = await
          contract.evaluateTransaction('GetAssetBidPrice',
          assetKey);
      const asset = JSON.parse(resultBuffer.toString('utf8'));
      console.log(`*** Result: GetAssetBidPrice,
          ${JSON.stringify(asset)}`);

   } catch (evalError) {
      console.log(`*** Falla: error evaluateTransaction
          GetAssetBidPrice: ${evalError}`);
   }
}

async function readSalePrice(assetKey, org, contract) {
   console.log(`${GREEN}--> Transaccion: GetAssetSalesPrice, -
       ${assetKey} desde organizacion ${org}${RESET}`);
   try {
      const resultBuffer = await
          contract.evaluateTransaction('GetAssetSalesPrice',
          assetKey);
```

```
        const asset = JSON.parse(resultBuffer.toString('utf8'));
        console.log(`*** Result: GetAssetSalesPrice,
            ${JSON.stringify(asset)}`);

    } catch (evalError) {
        console.log(`*** Falla: error evaluateTransaction
            GetAssetSalesPrice: ${evalError}`);
    }
}

function checkAsset(org, resultBuffer, ownerOrg) {
    let asset;
    if (resultBuffer) {
        asset = JSON.parse(resultBuffer.toString('utf8'));
    }

    if (asset) {
        if (asset.ownerOrg === ownerOrg) {
            console.log(`*** Result from ${org} - asset
                ${asset.assetID} owned by ${asset.ownerOrg}
                ReceiverID:${asset.receiverID}`);
        } else {
            console.log(`${RED}*** Falla: error chequeo de
                organizacion/institucion medica ${org} - asset
                ${asset.assetID} parte de ${asset.ownerOrg}
                ReceiverID:${asset.receiverID}${RESET}`);
```

```
      }
   }
}


async function readAssetByBothOrgs(assetKey, ownerOrg,
    contractOrg1, contractOrg2) {
   console.log(`${GREEN}--> Transaccions: ReadAsset, - ${assetKey}
       deberia ser perteneciente a ${ownerOrg}${RESET}`);
   let resultBuffer;
   resultBuffer = await
       contractOrg1.evaluateTransaction('ReadAsset', assetKey);
   checkAsset('Org1', resultBuffer, ownerOrg);
   resultBuffer = await
       contractOrg2.evaluateTransaction('ReadAsset', assetKey);
   checkAsset('Org2', resultBuffer, ownerOrg);
}


async function main() {
   console.log(`${GREEN} **** START ****${RESET}`);
   try {
      const randomNumber = Math.floor(Math.random() * 100) + 1;
      // cracion aleatoria de un asset
      const assetKey = `asset-${randomNumber}`;
```

```
/** ******* inicializacion cliente de Fabric: uso de
    identidad Org1 en Org1 Peer ******* */
const gatewayOrg1 = await initGatewayForOrg1();
const networkOrg1 = await gatewayOrg1.getNetwork(channelName);
const contractOrg1 = networkOrg1.getContract(chaincodeName);

/** ******* inicializacion cliente de Fabric: uso de
    identidad Org2 en Org2 Peer ******* */
const gatewayOrg2 = await initGatewayForOrg2();
const networkOrg2 = await gatewayOrg2.getNetwork(channelName);
const contractOrg2 = networkOrg2.getContract(chaincodeName);

try {
   let transaction;

   try {
   const ReceiverID = 'd1';
      const asset_properties = {
         object_type: 'asset_properties',
         asset_id: assetKey,
         ipfsHash:
             'Qmbq6Su7LzgYYgfQBzJUdXjgDUZZKxt4NSs4tbYwvfH8Wd',
         senderID: 'a1',
         receiverID: ReceiverID,
         salt:
             Buffer.from(randomNumber.toString()).toString('hex')
```

```
    };
    const asset_properties_string =
        JSON.stringify(asset_properties);
    console.log(`${GREEN}--> Submit Transaction:
        CreateAsset, ${assetKey} as Org1 - endorsed by
        Org1${RESET}`);
    console.log(`${asset_properties_string}`);
    transaction =
        contractOrg1.createTransaction('CreateAsset');
    transaction.setEndorsingOrganizations(org1);
    transaction.setTransient({
       asset_properties: Buffer.from(asset_properties_string)
    });
    await transaction.submit(assetKey, ReceiverID);
    console.log(`*** Result: committed, asset ${assetKey} is
        owned by Org1`);
} catch (createError) {
    console.log(`${RED}*** Falla: error : CreateAsset -
        ${createError}${RESET}`);
}

// lectura del asset desde ambas orgs
await readAssetByBothOrgs(assetKey, org1, contractOrg1,
    contractOrg2);
// Org1 deberia poder ver los registros privados
await readPrivateAsset(assetKey, org1, contractOrg1);
```

```
        // Org2 no deberia poder tener acceso a los privados
        await readPrivateAsset(assetKey, org2, contractOrg2);


      } catch (runError) {
        console.error(`Error en transaccion: ${runError}`);
        if (runError.stack) {
          console.error(runError.stack);
        }
        process.exit(1);
      } finally {
        // desconeccion
        console.log(`${GREEN}--> Cierra gateways`);
        gatewayOrg1.disconnect();
        gatewayOrg2.disconnect();
      }
    } catch (error) {
      console.error(`Error en setup: ${error}`);
      if (error.stack) {
        console.error(error.stack);
      }
      process.exit(1);
    }
    console.log(`${GREEN} **** FIN ****${RESET}`);
}
main();
```

5.3.2. Transferencia del asset

```
'use strict';


const { Gateway, Wallets } = require('fabric-network');
const FabricCAServices = require('fabric-ca-client');
const prompt = require('prompt-async');
const path = require('path');
const { buildCAClient, registerAndEnrollUser, enrollAdmin } =
    require('../../test-application/javascript/CAUtil.js');
const { buildCCPOrg1, buildCCPOrg2, buildWallet } =
    require('../../test-application/javascript/AppUtil.js');

const channelName = 'mychannel';
const chaincodeName = 'contract';

const org1 = 'Org1MSP';
const org2 = 'Org2MSP';
const Org1UserId = 'appUser1';
const Org2UserId = 'appUser2';

const RED = '\x1b[31m\n';
const GREEN = '\x1b[32m\n';
const RESET = '\x1b[0m';
```

```
const BLUE = "\x1b[34m\n"

async function initGatewayForOrg1() {
  console.log(`${GREEN}--> Inicio de GW y User: Usando Org1 con
      identidad de Org1 ${RESET}`);
  const ccpOrg1 = buildCCPOrg1();

  const caOrg1Client = buildCAClient(FabricCAServices, ccpOrg1,
      'ca.org1.example.com');

  const walletPathOrg1 = path.join(__dirname, 'wallet', 'org1');
  const walletOrg1 = await buildWallet(Wallets, walletPathOrg1);

  await enrollAdmin(caOrg1Client, walletOrg1, org1);
  await registerAndEnrollUser(caOrg1Client, walletOrg1, org1,
      Org1UserId, 'org1.department1');

  try {
    const gatewayOrg1 = new Gateway();
    await gatewayOrg1.connect(ccpOrg1,
      { wallet: walletOrg1, identity: Org1UserId, discovery: {
          enabled: true, asLocalhost: true } });

    return gatewayOrg1;
  } catch (error) {
    console.error(`Error coneccion con GW desde ORG1: ${error}`);
```

```
    process.exit(1);
  }
}

async function initGatewayForOrg2() {
  console.log(`${GREEN}--> Inicio de GW y User: Usando Org2 con
      identidad de Org2 Peer${RESET}`);
  const ccpOrg2 = buildCCPOrg2();
  const caOrg2Client = buildCAClient(FabricCAServices, ccpOrg2,
      'ca.org2.example.com');

  const walletPathOrg2 = path.join(__dirname, 'wallet', 'org2');
  const walletOrg2 = await buildWallet(Wallets, walletPathOrg2);

  await enrollAdmin(caOrg2Client, walletOrg2, org2);
  await registerAndEnrollUser(caOrg2Client, walletOrg2, org2,
      Org2UserId, 'org2.department1');

  try {
    const gatewayOrg2 = new Gateway();
    await gatewayOrg2.connect(ccpOrg2,
      { wallet: walletOrg2, identity: Org2UserId, discovery: {
          enabled: true, asLocalhost: true } });

    return gatewayOrg2;
  } catch (error) {
```

```
    console.error(‘Error in connecting to gateway for Org2:
        ${error}‘);
    process.exit(1);
  }
}

async function readPrivateAsset(assetKey, org, contract) {
  console.log(‘${GREEN}--> Evalua Transaccion:
      GetAssetPrivateProperties, - ${assetKey} from organization
      ${org}${RESET}‘);
  try {
    const resultBuffer = await
        contract.evaluateTransaction(’GetAssetPrivateProperties’,
        assetKey);
    const asset = JSON.parse(resultBuffer.toString(’utf8’));
    console.log(‘*** Result: GetAssetPrivateProperties,
        ${JSON.stringify(asset)}‘);

  } catch (evalError) {
    console.log(‘*** Falla evaluateTransaction readPrivateAsset:
        ${evalError}‘);
  }
}
async function getPrivateAsset(assetKey, org, contract) {
  console.log(‘${GREEN}--> Evalua Transaccion:
      GetAssetPrivateProperties, - ${assetKey} de la org
```

```
      ${org}${RESET}`);
  try {
    const resultBuffer = await
        contract.evaluateTransaction('GetAssetPrivateProperties',
        assetKey);
    const asset = JSON.parse(resultBuffer.toString('utf8'));
    return asset;
  } catch (evalError) {
    console.log(`*** Falla evaluateTransaction readPrivateAsset:
        ${evalError}`);
  }
}



function checkAsset(org, resultBuffer, ownerOrg) {
  let asset;
  if (resultBuffer) {
    asset = JSON.parse(resultBuffer.toString('utf8'));
  }

  if (asset) {
    if (asset.ownerOrg === ownerOrg) {
      console.log(`*** Result from ${org} - asset
          ${asset.assetID} parte de ${asset.ownerOrg}
          DESC:${asset.publicDescription}`);
```

```
    } else {
      console.log(`${RED}*** Falla owner check from ${org} -
          asset ${asset.assetID} parte de ${asset.ownerOrg}
          DESC:${asset.publicDescription}${RESET}`);
    }
  }
}


async function readAssetByBothOrgs(assetKey, ownerOrg,
    contractOrg1, contractOrg2) {
  console.log(`${GREEN}--> Evalua Transaccions: ReadAsset, -
      ${assetKey} debe ser parte ${ownerOrg}${RESET}`);
  let resultBuffer;
  resultBuffer = await
      contractOrg1.evaluateTransaction('ReadAsset', assetKey);
  checkAsset('Org1', resultBuffer, ownerOrg);
  resultBuffer = await
      contractOrg2.evaluateTransaction('ReadAsset', assetKey);
  checkAsset('Org2', resultBuffer, ownerOrg);
}


async function main() {
  console.log(`${GREEN} **** INICIO ****${RESET}`);
  try {
```

```
//creacion aleatoria
const randomNumber = Math.floor(Math.random() * 100) + 1;

/** ******* inicializacion cliente de Fabric: Org1 ******* */
const gatewayOrg1 = await initGatewayForOrg1();
const networkOrg1 = await gatewayOrg1.getNetwork(channelName);
const contractOrg1 = networkOrg1.getContract(chaincodeName);

/** ******* inicializacion cliente de Fabric: Org2 ******* */
const gatewayOrg2 = await initGatewayForOrg2();
const networkOrg2 = await gatewayOrg2.getNetwork(channelName);
const contractOrg2 = networkOrg2.getContract(chaincodeName);


prompt.start();
const {assetKey, pin} = await prompt.get([{name: "assetKey",
    description: 'Ingrese el asset ID', }, {name: 'pin',
    hidden: true, replace: '*'}]);

console.log(`Transfiriendo ${assetKey}`);

try {
  let transaction;
```

```
// lee los detalles publicos
await readAssetByBothOrgs(assetKey, org1, contractOrg1,
    contractOrg2);




try {
  // Agreement del Org1 con los datos provistos


  const asset_pin = {
    asset_id: assetKey,
    pin: pin,
    trade_id: randomNumber.toString()
  };
  const asset_pin_string = JSON.stringify(asset_pin);
  console.log(`${GREEN}--> Envia Transaccion:
      AgreeToTransfer, ${assetKey} como Org1${RESET}`);
  transaction =
      contractOrg1.createTransaction('AgreeToSell');
  transaction.setEndorsingOrganizations(org1);
  transaction.setTransient({
    asset_price: Buffer.from(asset_pin_string)
  });
```

```
  await transaction.submit(assetKey);
  console.log(`*** Resultado: aceptado, Org1 agreement
      completo ${assetKey}`);
} catch (sellError) {
  console.log(`${RED}*** Falla: AgreeToSell -
      ${sellError}${RESET}`);
}

await readPrivateAsset(assetKey, org1, contractOrg1);

await prompt.get([{description: "Enter para continuar",
    name: "enter"}]);


// las org se transfieren la informacion del asset
const assetJSON = await getPrivateAsset(assetKey, org1,
    contractOrg1);



console.log(`\n\n${RED} Forzando un error de PIN en Org2
    (Deberia Fallar)\n\n${RESET}`);
await prompt.get([{description: "Enter para continuar",
    name: "enter"}]);
```

```
try {
  // creacion del agreement con pin invalido = 0
  const asset_pin = {
    asset_id: assetKey,
    pin: 0,
    trade_id: randomNumber.toString()
  };
  const asset_pin_string = JSON.stringify(asset_pin);
  console.log(`${GREEN}--> Envia Transaccion: AgreeToBuy,
      ${assetKey} como Org2 ${RESET}`);
  transaction =
      contractOrg2.createTransaction('AgreeToBuy');
  transaction.setEndorsingOrganizations(org2);
  transaction.setTransient({
    asset_price: Buffer.from(asset_pin_string)
  });
  await transaction.submit(assetKey);
  console.log(`*** Resultado: agregado, ORG2 agrement para
      ${assetKey} con pin ${pin}`);
} catch (buyError) {
  console.log(`${RED}*** Falla: AgreeToBuy -
      ${buyError}${RESET}`);
}

try {
```

```
// Transferencia del Org1 al Org2
const asset_properties = {
  object_type: 'asset_properties',
  asset_id: assetKey,
  ipfsHash: assetJSON.ipfsHash,
  senderID: assetJSON.senderID,
  receiverID: assetJSON.receiverID,
  salt: assetJSON.salt
};
const asset_properties_string =
    JSON.stringify(asset_properties);
const asset_pin = {
  asset_id: assetKey,
  pin: pin,
  trade_id: randomNumber.toString()
};
const asset_pin_string = JSON.stringify(asset_pin);

console.log(`${GREEN}--> Envia Transaccion:
    TransferAsset, ${assetKey} como Org1${RESET}`);
console.log(`${asset_properties_string}`);
transaction =
    contractOrg1.createTransaction('TransferAsset');
transaction.setEndorsingOrganizations(org1, org2);
transaction.setTransient({
  asset_properties: Buffer.from(asset_properties_string),
```

```
    asset_price: Buffer.from(asset_pin_string)
  });
  await transaction.submit(assetKey, org2);
  console.log(`*** Resultado: , TransferAsset - Org2 tiene
      el asset ${assetKey}`);
} catch (transferError) {
  console.log(`${RED}*** Falla: TransferAsset -
      ${transferError}${RESET}`);
}



try {
  // creacion del agreement con pin valido
  const asset_pin = {
    asset_id: assetKey,
    pin: pin,
    trade_id: randomNumber.toString()
  };
  const asset_pin_string = JSON.stringify(asset_pin);
  console.log(`${GREEN}--> Envia Transaccion: AgreeToBuy,
      ${assetKey} como Org2${RESET}`);
  transaction =
      contractOrg2.createTransaction('AgreeToBuy');
  transaction.setEndorsingOrganizations(org2);
  transaction.setTransient({
```

```
        asset_price: Buffer.from(asset_pin_string)
      });
      await transaction.submit(assetKey);
      console.log(`*** Resultado: agregado, ORG2 agrement para
          ${assetKey} con pin ${pin}`);
    } catch (buyError) {
      console.log(`${RED}*** Falla: AgreeToBuy -
          ${buyError}${RESET}`);
    }


    // lee los detalles publicos
    await readAssetByBothOrgs(assetKey, org1, contractOrg1,
        contractOrg2);

    // Org1 deberia poder leer los detalles privados
    await readPrivateAsset(assetKey, org1, contractOrg1);
    // Org2 no deberia poder leer los detalles privados
    await readPrivateAsset(assetKey, org2, contractOrg2);



console.log(`\n\n${RED} Forzando una transferencia iniciada por
    Org2 a Org2 (Deberia Fallar)\n\n${RESET}`);
```

```
await prompt.get([{description: "Enter para continuar", name:
    "enter"}]);


      try {
        //Org2 tratando de "robar" el asset
        const asset_properties = {
          object_type: 'asset_properties',
          asset_id: assetKey,
          ipfsHash: assetJSON.ipfsHash,
          senderID: assetJSON.senderID,
          receiverID: assetJSON.receiverID,
          salt: assetJSON.salt
        };
        const asset_properties_string =
            JSON.stringify(asset_properties);
        const asset_pin = {
          asset_id: assetKey,
          pin: pin,
          trade_id: randomNumber.toString()
        };
        const asset_pin_string = JSON.stringify(asset_pin);

        console.log(`${GREEN}--> Envia Transaccion:
            TransferAsset, ${assetKey} como Org2 ${RESET}`);
        console.log(`${asset_properties_string}`);
```

```
  transaction =
      contractOrg2.createTransaction('TransferAsset');
  transaction.setEndorsingOrganizations(org1, org2);
  transaction.setTransient({
    asset_properties: Buffer.from(asset_properties_string),
    asset_price: Buffer.from(asset_pin_string)
  });
  await transaction.submit(assetKey, org2);
  console.log(`${RED}*** FALLA: se acepto a Org2
      ${assetKey}${RESET}`);
} catch (transferError) {
  console.log(`*** Exito: TransferAsset -
      ${transferError}`);
}


console.log(`${BLUE} Transfiriendo el asset desde Org1 a
    Org2, iniciado por Org1${RESET}`);
await prompt.get([{description: "Enter para continuar",
    name: "enter"}]);



try {
  // Transferencia del Org1 al Org2
  const asset_properties = {
```

```
  object_type: 'asset_properties',
  asset_id: assetKey,
  ipfsHash: assetJSON.ipfsHash,
  senderID: assetJSON.senderID,
  receiverID: assetJSON.receiverID,
  salt: assetJSON.salt
};
const asset_properties_string =
    JSON.stringify(asset_properties);
const asset_pin = {
  asset_id: assetKey,
  pin: pin,
  trade_id: randomNumber.toString()
};
const asset_pin_string = JSON.stringify(asset_pin);

console.log(`${GREEN}--> Envia Transaccion:
    TransferAsset, ${assetKey} as Org1 - endorsed by
    Org1${RESET}`);
console.log(`${asset_properties_string}`);
transaction =
    contractOrg1.createTransaction('TransferAsset');
transaction.setEndorsingOrganizations(org1, org2);
transaction.setTransient({
  asset_properties: Buffer.from(asset_properties_string),
  asset_price: Buffer.from(asset_pin_string)
```

```
  });
  await transaction.submit(assetKey, org2);
  console.log(`*** Results: committed, TransferAsset - Org2
      now owns the asset ${assetKey}`);
} catch (transferError) {
  console.log(`${RED}*** Falla: TransferAsset -
      ${transferError}${RESET}`);
}
await prompt.get([{description: "Enter para continuar",
    name: "enter"}]);

// lee los detalles publicos
await readAssetByBothOrgs(assetKey, org2, contractOrg1,
    contractOrg2);

// Org2 deberia poder leer los detalles privados
await readPrivateAsset(assetKey, org2, contractOrg2);
// Org1 no deberia poder leer los detalles privados
await readPrivateAsset(assetKey, org1, contractOrg1);


// lee los detalles publicos
await readAssetByBothOrgs(assetKey, org2, contractOrg1,
    contractOrg2);
```

```
  } catch (runError) {
    console.error(`Error en transaccion: ${runError}`);
    if (runError.stack) {
      console.error(runError.stack);
    }
    process.exit(1);
  } finally {
    // Desconectar
    console.log(`${GREEN}--> Cierra gateways`);
    gatewayOrg1.disconnect();
    gatewayOrg2.disconnect();
  }
} catch (error) {
  console.error(`Error en setup: ${error}`);
  if (error.stack) {
    console.error(error.stack);
  }
  process.exit(1);
```

```
    }
    console.log(`${GREEN} **** FIN ****${RESET}`);
  }
  main();
```

5.4. Aplicacion en Javascript para interactuar en blockchain PKI

```
  'use strict';

const { Gateway, Wallets } = require('fabric-network');
const FabricCAServices = require('fabric-ca-client');
const path = require('path');
const { buildCAClient, registerAndEnrollUser, enrollAdmin } =
    require('../../test-application/javascript/CAUtil.js');
const { buildCCPOrg1, buildWallet } =
    require('../../test-application/javascript/AppUtil.js');

const channelName = 'mychannel';
const chaincodeName = 'pki';
const mspOrg1 = 'Org1MSP';
const walletPath = path.join(__dirname, 'wallet');
const org1UserId = 'appUser_pki';

function prettyJSONString(inputString) {
```

```
    return JSON.stringify(JSON.parse(inputString), null, 2);
}
//comienzo
async function main() {
    try {
        // setea el connection profile
        const ccp = buildCCPOrg1();

        //instancia del CA
        const caClient = buildCAClient(FabricCAServices, ccp,
            'ca.org1.example.com');

        // creacion de wallet de usuarios
        const wallet = await buildWallet(Wallets, walletPath);

        // enrolamiento del admin
        await enrollAdmin(caClient, wallet, mspOrg1);

        // registracion de un usuario de la org1
        await registerAndEnrollUser(caClient, wallet, mspOrg1,
            org1UserId, 'org1.department1');

        // Creacion del gateway para las organizaciones
        const gateway = new Gateway();

        try {
```

```
await gateway.connect(ccp, {
   wallet,
   identity: org1UserId,
   discovery: { enabled: true, asLocalhost: true }
       //asLocalhost solo para uso de maqueta
});

// creacion de la instancia de red en el gateway creado
const network = await gateway.getNetwork(channelName);

// obtencion del contrato desde la red
const contract = network.getContract(chaincodeName);

// Se corre la funcion InitLedger del chaincode PKI
console.log('\n--> Transaccion: InitLedger, creacion de
    assets inicial');
await contract.submitTransaction('InitLedger');
console.log('*** Resultado: committed');

// Corre Get all Assets para revisar el resultado anterior
console.log('\n--> Transaccion: GetAllAssets, Trae todos
    los assets');
let result = await
    contract.evaluateTransaction('GetAllAssets');
```

```
console.log(`*** Resultado:
    ${prettyJSONString(result.toString())}`);

// Corre AssetExists para revisar el resultado anterior

console.log('\n--> Transaccion: AssetExists, funcion
    devuelve verdadero si existe');
result = await contract.evaluateTransaction('AssetExists',
    'asset1');
console.log(`*** Resultado:
    ${prettyJSONString(result.toString())}`);


// Corre ReadAsset para revisar el resultado anterior

console.log('\n--> Transaccion: ReadAsset, funcion trae el
    asset1');
result = await contract.evaluateTransaction('ReadAsset',
    'asset1');
console.log(`*** Resultado:
    ${prettyJSONString(result.toString())}`);

try {
//Prueba forzando un error (trata de actualizar un asset
    no existente)
```

```
        console.log('\n--> Transaccion: UpdateAsset actualiza el
            asset70 que no existe aun');
        await contract.submitTransaction('UpdateAsset',
            'asset70', 'PK XXXX', 'Juan Perez', 'UNSAM');
        console.log('******** ERROR');
    } catch (error) {
        console.log(`*** Error Capturado: \n ${error}`);
    }
    // TransferAsset cambia la org del medico 1 a UCA desde UBA
    console.log('\n--> Transaccion: TransferAsset cambia el
        org a UCA');
    await contract.submitTransaction('TransferAsset',
        'asset1', 'UCA');
    console.log('*** Resultado: correcto');

    // Corre ReadAsset para revisar el resultado anterior

    console.log('\n--> Transaccion: ReadAsset');
    result = await contract.evaluateTransaction('ReadAsset',
        'asset1');
    console.log(`*** Resultado:
        ${prettyJSONString(result.toString())}`);
} finally {
    // Disconnect from the gateway when the application is
        closing
    // This will close all connections to the network
```

```
        gateway.disconnect();
      }
   } catch (error) {
      console.error(`******** FALLA en correr la app: ${error}`);
   }
}

main();
```

Indice Alfabetico

Referencias

[1] M. de Salud, "La estrategia nacional de salud digital," https://www.argentina.gob.ar/salud/digital.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf."

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, and et al., "Hyperledger fabric," *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[4] A. Bessani, E. Alchieri, J. Sousa, A. Oliveira, and F. Pedone, "From byzantine replication to blockchain: Consensus is only the beginning," 2020.

[5] T. H. A. W. Group, "Hyperledger architecture, volume i introduction to hyperledger business blockchain design philosophy and consensus," *Hyperledger*, 2019.

[6] D. Torre and S. Seang, "Proof of work and proof of stake consensus protocols: a blockchain application for local complementary currencies," *HAL CCSD*, 2019.

[7] T. H. A. W. Group, "Hyperledger architecture, volume ii smart contracts," *Hyperledger*, 2019.

[8] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm (extended version)," *Stanford University*, 2014.

[9] 1, "Ipfs - content addressed, versioned, p2p file system," *IPFS*, 2014.

[10] M. David and M. Petar, "Kademlia: A peer-to-peer information system based on the xor metric," *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, 2002.

[11] B. Ingmar and M. Sebastian, "A practicable approach towards secure key-based routing," *IEEE*, 2008.

[12] M. J. Freedman, E. Freudenthal, and D. Mazieres, "Democratizing content publication with coral," *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, 2004.

[13] V. Almada, A. Carratala, H. Scolnik, and otros, "http://servicios.infoleg.gob.ar/infoleginternet/anexos/40000-44999/42392/norma.htm," *ADMINISTRACION PUBLICA NACIONAL*, 1997.