laSalle
UNIVERSITAT RAMON LLULL

Escola Tècnica Superior d'Enginyeria Electrònica i Informàtica La Salle

Treball Final de Màster

Máster Universitario en Ingenieria de Telecomunicaciones

Tecnología Blockchain, la nueva era de los Contratos Comerciales

Nombre Alumno	Nombre Profesor Ponente
Laura Esbri Vidal	Julia Sánchez Rodrigez

ACTA DEL EXAMEN
DEL TRABAJO FINAL DE MÁSTER

Reunido el Tribunal calificador en la fecha indicada, el alumno

D. Laura Esbri Vidal

expuso su Trabajo Final de Máster, titulado:

Tecnología Blockchain, la nueva era de los Contratos Comerciales

Acabada la exposición y contestadas por parte del alumno las objeciones formuladas por los Sres. miembros del tribunal, éste valoró dicho Trabajo con la calificación de

Barcelona,

VOCAL DEL TRIBUNAL VOCAL DEL TRIBUNAL

PRESIDENTE DEL TRIBUNAL

Abstract

The Master's Thesis aims to design and implement a blockchain-based tool that optimises the commercial process between two different companies. The current way of carrying out the process is inefficient and at the end of the year it is reflected in the form of losses.

The present document details the birth of the blockchain technology with the Distributed Ledger Technologies, makes the study of new concepts introduced due to this new technology, as well as the different implementations that exist nowadays. In addition to highlighting the benefits that blockchain technology has to bring to other sectors, not only the banking sector.

The novelty of the blockchain network that will be designed and implemented in this project is the version 2.0 of Hyperldger Fabric adapted for Kubernetes. This is one of the first blockchain networks in this version to be deployed in a Cloud Banking environment, meeting all security requirements.

To conclude the current document, a complete installation guide for the Hyperldger Fabric network is presented with the results and conclusions.

Palabras Clave: Blockchain, Hyperledger Fabric, business process, optimization, traceability, immutability, kubernetes, docker, cloud, banking sector.

Resumen

El trabajo de final de Máster tiene como objetivo el diseño e implementación de una herramienta basada en *blockchain* que optimice el proceso comercial entre dos empresas diferentes. El actual modo de realizar el proceso es ineficiente y al cabo del año se ve reflejado en forma de pérdidas.
El presente documento detalla el nacimiento de la tecnología *blockchain* con las *Distributed Ledger Technologies,* realiza el estudio de nuevos conceptos introducidos debido a esta nueva tecnología, así como las diferentes implementaciones que existen actualmente. Además de remarcar los beneficios que tiene que aportar la tecnología blockchain a otros sectores, no solo al bancario.
Lo novedoso de la red *blockchain* que se va a diseñar e implementar en este proyecto es la versión 2.0 de *Hyperldger Fabric* adaptada para *Kubernetes*. Nos encontramos delante de una de las primeras redes *blockchain* de esta versión en ser desplegadas en un entorno *Cloud* Bancario cumpliendo con los requisitos de seguridad.
Para concluir con el documento actual se presenta una guía de instalación completa de la red *Hyperldger Fabric* con los resultados y las conclusiones.

Palabras Clave: Blockchain, Hyperledger Fabric, Proceso comercial, optimización, trazabilidad, inmutabilidad, kubernetes, docker, cloud, sector bancario.

Índice general

Índice figuras

Índice de tablas

Acrónimos

ACL: *Access Control List.*
API: Interfaz de programación de aplicaciones.
BaaS: *blockchain as a Service.*
BBDD: Bases de datos.
BFT: *Byzantine Fault tolerance*.
CaaS: *Container as a Service.*
CA: *Certificate Authority.*
CCID: *Chaincode Identifier.*
CDF: *Crash Default Tolerance.*
CLI: *Command Line Interface.*
CPD: Centro de procesamiento de datos.
CPU: *Central Processing Unit.*
CXB: Caixabank.
DinD: *Docker-in-Docker.*
DLT: *Distributed Ledger Technology.*
DSM: *Digital Single Market.*
FQDN: *Fully Qualified Domine Name.*
GDPR: *General Data Protection Regulation.*
HTTP: *Hypertext Transfer Protocol.*
JSON: *JavaScript Object Notation.*
MSP: Membership Service Provider.
NFS: *Network File System.*
P2P: *Peer-to-Peer.*
PBFT: *Practical Byzantine Fault Tolerance.*
Poc: *Proof of Concept.*
PoS: *Proof of Stake.*
PoW: *Proof of Work.*
PV: *Persistent Volumes.*
PVC: *Persistent Volume Claim.*
REST: *Representational State Transfer.*
SDK: *Software Development Kit.*

Capítulo 1: Introducción

La irrupción de Internet ha cambiado radicalmente nuestra forma de vida y la sociedad como la conocíamos. La evolución a nivel mundial y el crecimiento exponencial que ha realizado durante las últimas dos décadas ha hecho posible la creación de nuevas industrias y modelos de negocio en sectores de la economía consolidados durante siglos.

Internet ha sido uno de los inventos que ha marcado un antes y un después en la historia de la humanidad, situándose al mismo nivel que la invención de la escritura, la imprenta o la electricidad. En la actualidad, hablamos del concepto "Internet de la información" como resultado de ser el medio de información, difusión y comunicación más amplio que ha existido desde el principio de la historia. Podemos acceder, replicar y almacenar libremente cualquier tipo de dato sin saber el origen de la fuente de forma fehaciente y sin tener la certeza de la privacidad de nuestros datos. No obstante, su crecimiento se ha realizado de forma centralizada, donde cuatro grandes compañías tecnológicas (Google, Amazon, Facebook, Apple) monopolizan la actividad que se genera en Internet.

La tecnología *blockchain* define el próximo paso en la evolución natural de la red, aportando un nuevo concepto el "Internet del valor". Donde se intercambiará el valor de forma descentralizada, sin la necesidad de una entidad de confianza que intermedie, realice y confirme las interacciones entre los participantes. No se trata de sustituir el actual Internet, sino de crear una capa adicional en la que las personas puedan intercambiar valor entre ellas. De esta forma, Internet revolucionó el acceso y la difusión de la información, mientras que la *blockchain* supone la revolución en la transmisión y el valor de los datos [1]. Esa capacidad es lo que convierte a la tecnología *blockchain* en algo que puede revolucionar nuestra forma de entender el mundo.

La directora de Desarrollo Estratégico e Innovación de Caixabank, Mariona Vicens (2017) augura que la tecnología *blockchain* tendrá dos incidencias principales: "La primera se encuentra en las operaciones donde todavía hay intermediarios o se tienen que hacer muchas comprobaciones manuales". Y continúa: "También vemos que influirá a la hora de ayudar a los reguladores y a las instituciones a ser más transparentes" [2].

Es importante considerar que sectores como la banca y las finanzas, desde la crisis financiera global de 2008, experimentan dificultades para restablecer los niveles de rentabilidad, solvencia y, sobre todo, recuperar la confianza perdida de los clientes [3]. El sector bancario ha querido revertir estas tendencias aumentando su inversión en tecnologías de la información como *Cloud Computing, blockchain,* Inteligencia Artificial o *Quantum Computing*, entre otros.

El contexto en el que se inicia el presente proyecto de final de máster se sitúa hace dos años en el departamento de Innovación de la Empresa *ITnow*.
ITnow es una *joint venture* entre 'la Caixa' e *IBM* que proporciona servicios de Infraestructura de Tecnología de la Información (TI) al grupo "la Caixa". Fue fundada en enero de 2012. Los servicios principales proporcionados incluyen centro de datos, mainframe, plataforma final de cliente, telecomunicaciones y toda la infraestructura

tecnológica necesaria para proporcionar servicios eficientes a *CaixaBank* y todas sus filiales, llamadas también empresas del 'Grupo la Caixa'.
El departamento de Innovación de *ITnow* participa directamente con el Centro de Innovación Digital de *IBM*, investigando y poniendo a prueba nuevas tecnologías que permitirán a *CaixaBank* impulsar su negocio y consolidarla como una entidad líder en el sector bancario [4].

1.1. Objetivos del proyecto

El trabajo de Final de Máster tiene la finalidad de dar a conocer la Tecnología de Libro Mayor Distribuido, específicamente, una de sus implementaciones: la *Blockchain*. Por lo tanto, en este proyecto se analizan las características genéricas de esta tecnología, así como los componentes que la definen y las aplicaciones más relevantes para el mundo de la Tecnología de la Información. Asimismo, se estudian las características, los componentes y la arquitectura de una implementación de *blockchain* en concreto *Hyperledger Fabric*, la tecnología *blockchain*, estableciendo el marco teórico sobre las cadenas de bloques y sus principales características.
Para ello es necesario cumplir los siguientes objetivos:

- Establecer el contexto teórico sobre el inicio de los sistemas distribuidos, las implementaciones DLT (Distributed Ledger Technologies), más concretamente la tecnología blockchain.
- Analizar las características y elementos inherentes de las cadenas de bloques, así como las ventajas y las desventajas que presentan.
- Estudiar las diferentes tipologías e implementaciones que existen, como también las aplicaciones más destacadas.
- Conocer en profundidad el proyecto Hyperledger Fabric, sus componentes y su funcionalidad.
- Diseñar e implementar un entorno Cloud de pruebas lo más parecido al entorno Cloud corporativo.
- Diseñar, gestionar e implementar una prueba de concepto (PoC) basada en Hyperledger Fabric que permita la optimización de un proceso comercial.
- Gestionar el desarrollo de la API-Rest que se comunica con la red blockchain y el front-end.
- Gestionar el desarrollo del Smart Contract con la lógica de negocio del proceso comercial.
- Desarrollar, optimizar y securizar la arquitectura Cloud de Hyperledger Fabric para el despliegue en producción, según la normativa de la empresa, así como todos los procesos de monitorización y mantenimiento necesarios para la operación de la aplicación.
- Demostración y conclusiones del proyecto.

1.2. Motivación

El departamento de Innovación de *ITnow* tiene el objetivo de hacer eficiente procesos, aplicaciones y metodologías, entre otros, mediante la implantación de tecnologías que están en fase de desarrollo o que aún no han sido implantadas en el sector bancario o financiero.

El objetivo común que existe entre *ITnow* e *IBM* es conseguir que *Caixabank* sea puntera en investigación, innovación y desarrollo de proyectos para mejorar los procesos internos.

La motivación de este proyecto de final de Máster nace del departamento de mejora contínua de *ITnow* que se encarga de realizar estudios de eficiencia a varios niveles. Durante años este departamento ha encontrado una serie de déficits en la gestión de las adendas comerciales entre dos filiales de *Caixabank*. En el proyecto estas dos filiales serán llamadas A y B debido a la confidencialidad.

La mayoría de las veces el departamento de mejora continua se pone en contacto con el departamento de innovación de *ITnow* para proponer pruebas de concepto que aporten un valor añadido a las empresas del 'Grupo la Caixa'.

En este trabajo de final de Máster se llevará a cabo el desarrollo de una herramienta de uso interno basada en tecnología *blockchain* para el desempeño de un proceso comercial. En el apartado 4.2 se define el proceso comercial.

Capítulo 2: Estado del Arte

En el presente capítulo se contextualiza al lector con el contexto técnico de la tecnología blockchain, así como sus características inherentes y ventajas. Se va a exponer las principales implementaciones, pero en especial Hyperledger Fabric.

2.1. Contexto Técnico

El bloque génesis, el primer bloque de la cadena de *Bitcoin* fue creado el 3 de enero de 2009, se trata de la primera implementación de una criptomoneda basada en la tecnología *blockchain*. La teoría de la cadena de bloques es la consecuencia de más de 40 años de investigación en el mundo de la criptografía [5]. La criptografía es esencial en *blockchain* debido a que la información que viaja dentro de la red de servidores es encriptada.

A mediados del siglo XX, los avances matemáticos hacen posible el desarrollo de la criptografía de clave pública y privada. Esto sentó las bases teóricas para que Ralph Merkle presentara la idea de encadenar de forma inmutable bloques de información con una función hash criptográfica, recibiendo el nombre de *Merkle hash tree*. En el estudio publicado en 1978, el creador explica cómo se puede vincular información siguiendo una estructura de árbol. Otro avance decisivo fue la invención del algoritmo RSA, los investigadores Rivest, Shamir y Adleman presentaron el sistema criptográfico de clave pública que se sigue utilizando para la generación de claves, el cifrado y el descifrado de mensajes.

Las bases técnicas de la criptografía establecieron las principales características de la *blockchain*. El artículo presentado en el año 1982 por el matemático e informático teórico David Chaum [6], describe el diseño de un sistema informático distribuido que puede ser establecido, mantenido y confiable por grupos que desconfían mutuamente. Chaum en el mismo artículo también propuso el primer protocolo de pagos digitales basado en la primitiva criptográfica de la firma ciega.

El estudio de la criptografía y de los sistemas descentralizados se intensificó debido a un movimiento que surgió en los años 90, que buscaba defender la privacidad y el acceso a la información junto a la libertad de expresión. Este movimiento desencadenó la creación de protocolos y tecnologías que facilitaron la publicación del estudio *Bitcoin* P2P *e-cash* el 31 de octubre de 2008 por Satoshi Nakamoto. El estudio publicado describe un sistema de dinero electrónico *peer-to-peer* sin la necesidad de intermediarios, para que meses más tarde, con la generación del bloque génesis, se iniciara la red *Bitcoin[7]*.

El proceso histórico comentado sirve para introducir al lector en el contexto técnico de las tecnologías del presente documento, y es necesario para remarcar la existencia de diferentes protocolos descentralizados. La gran mayoría están centrados en generar

protocolos de dinero digital y, los que vamos a tratar en este proyecto, están orientados en almacenar datos de forma distribuida y segura.

2.2. Distributed Ledger Technology

La Tecnología de Libro Mayor Distribuido o, en inglés, *Distributed Ledger Technology* es una tecnología que mantiene y gestiona el registro de las transacciones que realizan los participantes de la red. El libro de cuentas o *ledger* contiene la información que se almacena de forma descentralizada en los nodos de la red.

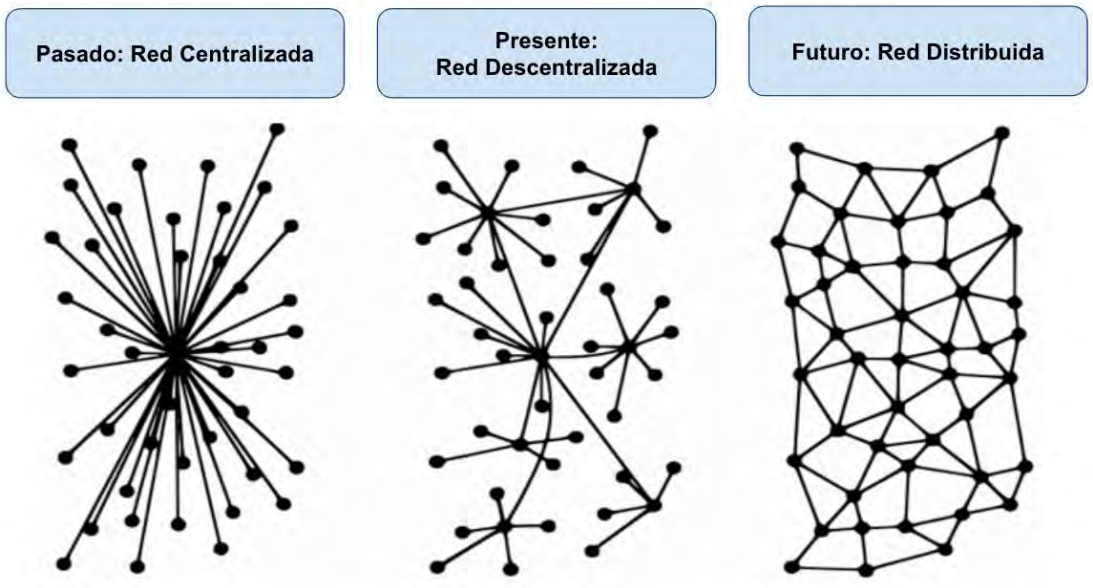


Figura 1. Red centralizada, descentralizada y distribuida.

El concepto principal de esta tecnología se rige por ser independiente de una entidad central que valide las transacciones, y por ello, no es necesaria la existencia de una autoridad que regule el funcionamiento de la red.

No obstante, al tratarse de un entorno de desconfianza, se requiere de algoritmos de consenso para garantizar la integridad y la veracidad de la información almacenada, de esta forma los nodos de la red validan los nuevos datos que van a formar parte de la DLT.

La implementación más conocida hoy en día de esta tecnología es la *blockchain* y la primera de ellas fue *Bitcoin*. Existen otras implementaciones de la tecnología de registro distribuido como por ejemplo *Tangle de IOTA* [8] *o HashGraph* [9] de *Leemon Baird*. En el mundo de las finanzas, frecuentemente se emplean los conceptos DLT y *blockchain* como sinónimos, hay que mantener en claro que una blockchain es un tipo de DLT, y existen DLTs que no cumplen las características de una *blockchain*.

2.3. Blockchain

Bockchain o cadena de bloques es una estructura de datos distribuida que es replicada y compartida por los nodos que componen la red. Se trata de una red distribuida *peer-to-peer* (P2P) debido a que no existe una entidad o autoridad central que actúa como proveedora de confianza o validadora única de las transacciones que se realizan. El libro mayor distribuido (*ledger*) contiene el registro de todas las transacciones realizadas en la red. La información de las transacciones es securizada criptográficamente y es almacenada en forma de bloque. Los bloques al unirse enlazan su propia información con la existente, formando una cadena inalterable. Las transacciones solo se pueden agregar a la cadena de bloques en orden secuencial de tiempo.
Las anteriores propiedades garantizan que, una vez se han agregado los datos en la cadena de bloques, es posible recuperar la información original almacenada debido a que los datos no pueden ser modificados, considerándose así prácticamente inmutables.
La integridad se obtiene ya que, una posible alteración del contenido de una transacción necesita alterar cada uno de los bloques de la cadena para ocultar dicha información.

2.3.1. Componentes principales de una red blockchain

Los componentes que forman parte de una red *blockchain* son los siguientes:

- Los **nodos** son servidores que componen y participan activamente en una red blockchain. La red *blockchain* sigue una estructura *peer-to-peer*, esta facilita la comunicación y el intercambio directo de información. Este tipo de redes no necesitan una autoridad central que facilite la comunicación ya que actúan simultáneamente como clientes y servidores respecto a los demás nodos de la red, comportándose como iguales.

- Los **participantes** de una red *blockchain* pueden estar formados por un único nodo *peer* o hasta un conjunto de ellos, representando empresas, organizaciones y consorcios.

- La **transacción** es definida como la unidad fundamental de *blockchain*. Los nodos de una red *blockchain* se comunican mediante transacciones. Las transacciones son transferencias de valor o de activos entre nodos.

- Los **activos** o, en inglés, *assets* son un conjunto de transacciones agrupadas que representan un bien, un servicio o una propiedad tangible o intangible que se encuentran almacenadas o vinculadas en la red *blockchain*.

- Los **bloques** contienen un conjunto ordenado de transacciones. La *blockchain* se define como una lista vinculada de bloques. Los bloques presentan dos piezas importantes la cabecera y los datos. La cabecera contiene el número del bloque, la copia del hash del bloque anterior y el hash del bloque actual. Los datos contienen una lista de transacciones donde se detalla el activo, así como el emisor, el receptor y el

timestamp. La lista de transacciones se sintetiza siguiendo la función *hash merkle tree*. El primer bloque de la cadena se llama bloque génesis [10].

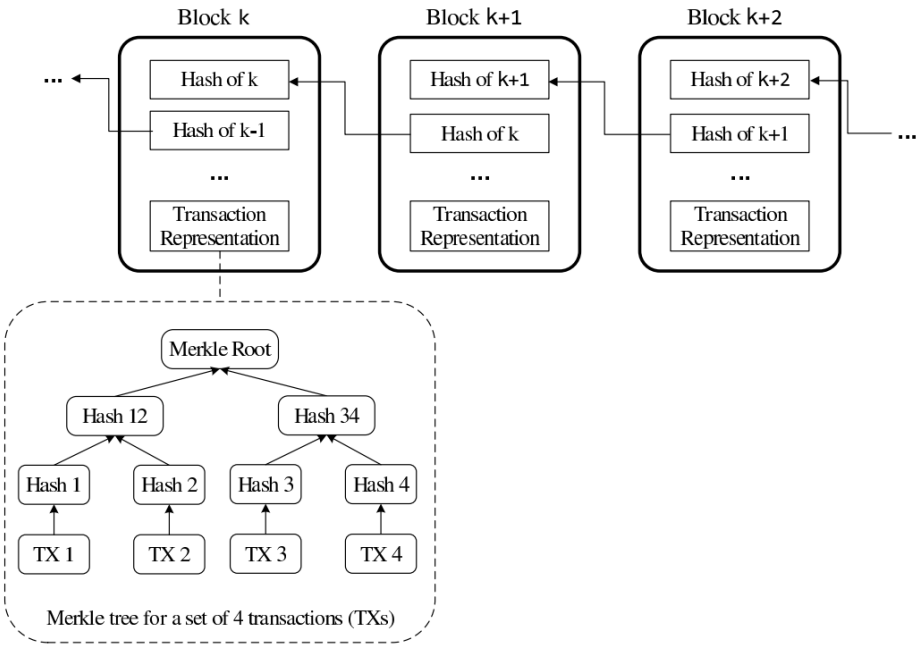


Figura 2. Bloque y cadena de bloques. La ilustración de una cadena de bloques con la estructura de estos. La lista de transacciones se encripta siguiendo el árbol de Merkle [11].

- La ***ledger*, libro mayor distribuido, o libro de cuentas compartido** es el registro de todas las operaciones y transacciones que realizan los nodos dentro de la blockchain. El libro de cuentas, que contiene la información, se encuentra replicado exactamente en cada uno de los nodos, por lo que está distribuido en toda la red.

- El ***Smart Contract*** [12] es un programa informático en el que está implementado una serie de acuerdos previamente definidos, establecidos y pactados entre dos o más partes. Los contratos inteligentes se ejecutan una capa por encima de la *blockchain*, siendo un programa no controlado por ninguna de las partes de la red. Se ejecutan de forma automática y sin intermediarios cuando se cumplen las condiciones con las que se ha programado y han acordado mutuamente las partes que participan en la red. El resultado de la ejecución queda escrito de manera inmutable y transparente.

- El **consenso** es el conjunto de algoritmos que garantizan que todos los participantes llegan a un acuerdo para validar las transacciones que van a formar parte de la red *blockchain*.

2.3.2. Principales características

La tecnología *blockchain* tiene notables beneficios que vienen de sus principales características:

- Descentralización: Se elimina la necesidad de tener un tercero de confianza o intermediario que valide las transacciones. En su lugar, *blockchain* proporciona el mecanismo de consenso, aportando así, la confianza en la red.

- Trazabilidad: todos los registros y transacciones que se realizan dentro de la red, tanto los que se validan como los que son rechazados, quedan almacenados en la *ledger*. Esto es debido a la naturaleza de la red de a*ppend-only*.

- Transparencia: es debida a que cada transacción de información es replicada y compartida con cada uno de los nodos que forman la red *blockchain*. De esta forma, todos los nodos observan las mismas acciones, convirtiendo el sistema en transparente.

- Inmutabilidad y seguridad: la forma de almacenar la información en bloques que se entrelazan criptográficamente consigue que la información no pueda ser modificada sin ser alterada, proporcionando integridad en la información y seguridad en la red.

- Alta disponibilidad: la *blockchain* es un sistema formado por un gran número de nodos, los datos son replicados y actualizados en cada nodo. La información se encuentra disponible en cualquier momento, a pesar de que algunos nodos desaparezcan o queden inaccesibles, la red continúa funcionando. La redundancia de las transacciones la hace tolerante a fallos.

- Simplificación de procesos e intercambio de datos: la gran mayoría de las entidades mantiene cada una su propio sistema provocando incompatibilidades y dificultades a la hora de integrar sistemas y procesos con otras empresas. *blockchain* facilita la transmisión de información entre las partes interesadas. También reduce la complejidad de procesos que pueden ser verificados y aprobados por la red, de forma automática sin la necesidad de supervisión humana.

- Ahorro de costes: la automatización de los procesos, la eliminación de un tercero de confianza que valide las transacciones, la agilización del envío de información y la inherente seguridad y trazabilidad que van ligadas a la tecnología *blockchain* facilitan el ahorro de costes.

2.3.3. Desventajas de la tecnología

Todas las tecnologías presentan algunos desafíos que hay que abordar para tener un sistema robusto a fallos, práctico y accesible para la mayoría.
Las redes blockchain son principalmente sensibles a lo siguiente:

La **escalabilidad** es la capacidad que tiene un sistema de crecer en tamaño y gestionar el aumento de carga de trabajo. Cuando un sistema es poco escalable necesita soluciones y esfuerzos adicionales que modifican el sistema para hacer frente al aumento de peticiones.
La escalabilidad de un sistema basado en tecnología *blockchain* hace referencia a la cantidad de transacciones que la red puede procesar cuando aumenta el número de participantes y de transacciones.
La primera causa es la descentralización, como se ha explicado antes en este documento, para llegar al consenso, todos los nodos de la red deben procesar todas las transacciones. En consecuencia, el límite de procesamiento de la red es el límite del nodo con menos recursos de la red.
Por otro lado, si se aumenta el número de nodos, se aumenta el tiempo necesario para alcanzar el consenso en toda la red.
La segunda causa es el tamaño de los bloques, el cual se encuentra limitado para evitar que aumente considerablemente la propagación del bloque por la red. Una vez finalizada y aceptada una transacción por todos los nodos, se genera un bloque que se distribuye por la red. Cuanto mayor sea el bloque, más tardará en transmitirse y almacenarse en todos los nodos. Se puede dar el caso de que un bloque se genere sin que el anterior haya llegado a todos los nodos y sin que haya sido guardado en todas las *ledgers*. Esto provoca la existencia de bloques huérfanos en la red y bifurcaciones de la ledger, en inglés *forks*. Las bifurcaciones se originan cuando dos bloques diferentes tienen la misma posición en diferentes nodos, ocasionando dos *ledgers* diferentes en una misma red.
Para evitar este último problema, en el momento de generación del bloque se aplica un tiempo mínimo. De esta forma, se deja el tiempo suficiente para que el anterior bloque se transmita por toda la red y se almacene en los nodos.
Por lo que las redes *blockchain*, en cuanto a escalabilidad, sufren limitaciones en el número de nodos, en los recursos de procesamiento del nodo y la información que se transmite en los bloques es limitada.

La **adaptabilidad** de un sistema informático es la capacidad de ajustarse de forma eficiente y sencilla a los procesos y necesidades organizativas del entorno donde va a ser desplegado. En otras palabras, es la habilidad del sistema de ser customizable según las exigencias de diseño sin que se vean afectadas las funcionalidades ni las características inherentes del sistema.

En este sentido, *blockchain* ha tratado de abordar esta problemática con un gran abanico de protocolos que se adaptan según la aplicación final que se le quiera dar a esta tecnología. Sin embargo, la gran mayoría de los centros de procesamiento de datos (CPD) empresariales no están preparados para implantar y gestionar la arquitectura tecnológica necesaria derivada de implementar una *blockchain*.
En consecuencia, la necesidad empresarial de incorporar *blockchain* en nuevas aplicaciones y las dificultades reales para hacerlo, han originado compañías tecnológicas

que ofrecen el servicio de *blockchain* as a Service (BaaS). De esta forma, las empresas interesadas en el desarrollo de soluciones blockchain pueden contratar el servicio de forma externa.

La **regulación** [13] a nivel europeo está dirigida por la Comisión Europea que desde el 2017 trata de reforzar la cooperación y las inversiones en el despliegue de aplicaciones basadas en *blockchain*. Para ello, se ha centrado en apoyar el establecimiento de normas internacionales y facilitar el diálogo entre las partes interesadas de la industria y los reguladores. La promoción de un marco legal habilitante para el Mercado Único Digital (DSM) impulsado por la Comisión Europea lanzó un estudio sobre los aspectos legales, de gobernanza e interoperabilidad de las cadenas de bloques [14]. Fundamentalmente para potenciar dos áreas relacionadas con *blockchain* que podrían beneficiarse de la estandarización y la regulación jurídica. Estas dos áreas son los Contratos Inteligentes (Smart Contracts) y la tokenización de activos.

Los estudios enfocados en los *Smart Contracts* tratan de averiguar si la actual legislación está suficientemente definida para garantizar la exigibilidad de los contratos inteligentes. En caso de disputas legales tener las directrices marcadas para actuar en consecuencia.
Por otro lado, los estudios del marco legal actual de la tokenización tratan de evaluar si es correcto y apropiado para escenarios de comercialización y emisión de tokens cuando no se consideran instrumentos financieros.

Por otro parte, la información almacenada en las DLT y en las blockchain también está sujeta a legislaciones.

La **privacidad** de la información almacenada, como se ha comentado previamente, debe aplicar la ley de protección de datos europea (GDPR) [15]. Los derechos de la privacidad en el uso de los datos de carácter personal, el derecho a la rectificación o el derecho al olvido, junto con las características de descentralización y almacenamiento de los datos dificultan el desarrollo de los casos de uso.
En consecuencia, desde los inicios del diseño de una aplicación basada en *blockchain*, la privacidad y el cumplimiento de normativas marca su arquitectura y características de acceso a la red y a la información por los participantes.

La EU blockchain Observatory and Forum es una iniciativa de la Comisión Europea para acelerar la innovación de la *blockchain* y el desarrollo dentro de la unión europea. En esta iniciativa se divulgan una serie de documentos e informes académicos donde se están estandarizando los códigos de buenas prácticas relacionados con la tecnología *blockchain*. En el informe "*Conclusions and Reflections*" [16] hace una serie de recomendaciones para evitar vulnerabilidades y mantener la privacidad. Entre ellas, se encuentran las siguientes:

- Evitar almacenar datos de carácter personal en la *blockchain*. Para ello recomienda la anonimización de los datos mediante el uso de técnicas de cifrado, ofuscación y protocolos de reconocimiento cero ZKP (*Zero Knowledge proof*).
- Expone las posibles vulnerabilidades derivadas de errores de programación de los *smart contracts*, así como también, vulnerabilidades en la gestión y custodia de las claves criptográficas de acceso del usuario.

Con los dos últimos puntos del documento se ha demostrado que *blockchain* es una tecnología con una proyección prometedora, pero se encuentra en construcción y presenta retos significativos en el ámbito tecnológico y jurídico.

El elevado coste de los proyectos, junto con la escasez de regulación, provoca que muchas empresas no apuesten por la tecnología blockchain y se retracten hasta no tener un marco regulatorio en condiciones. Esto ha aumentado el interés por los Sandbox regulatorios. Los Sandbox regulatorios [17], en el ámbito de las finanzas y de la economía digital, son un entorno de pruebas para nuevos modelos de negocio que aún no están protegidos por una regulación vigente ni supervisado por instituciones regulatorias. De esta forma se combate a nivel empresarial la falta de estandarización en la tecnológica y legislativa.

2.3.4. Tipos de Blockchain

La tecnología *blockchain* ha evolucionado en los últimos años para cubrir las diversas necesidades de cada sector, manteniendo la seguridad, la eficiencia y la transparencia que la definen. En función de la problemática que quieren abordar, las redes *blockchain* se clasifican en función de las siguientes categorías:

- Accesibilidad
- Permisividad
- Descentralización

La categoría principal que se tiene en cuenta cuando se menciona esta tecnología es según el tipo de **acceso**:

- Las redes **públicas** o no permisionadas son de acceso público, no tienen restricciones de lectura de las transacciones. Tampoco existen restricciones sobre la participación, es decir, cualquier nodo puede participar en el proceso de toma de decisiones. En función de la red, los usuarios pueden ser recompensados, o no, por su participación. Todos los nodos mantienen una copia de la *ledger* y para actualizar el estado utilizan un mecanismo de consenso distribuido, por ejemplo, Bitcoin[7] o Ethereum [18]. Las cadenas con este tipo de acceso suelen tener consensos que implican la recompensa mediante tokens o criptomonedas.

- Las redes **privadas** son de acceso restringido, requiere que los participantes sean invitados a formar parte de la red. A parte, se pueden aplicar restricciones de lectura de las transacciones y de participación. El acceso a las redes *blockchain* privadas se realiza mediante autenticación otorgada por una organización o un conjunto de participantes que adoptan el rol de administrador de red, o como es el caso de Hyperledger Fabric [19], por listas de control de acceso (ACL). Estas cadenas de bloques, debido a su consenso, no necesitan tener criptomonedas o tokens.

- Las redes **federadas** o **consorcios**, también llamadas **híbridas**, son redes formadas por cadenas de bloques públicas y privadas. Hay varias implementaciones de este tipo de *blockchain* ya que suelen evitar los principales inconvenientes que

tienen las blockchain públicas y privadas sin renunciar a la descentralización, la escalabilidad o a la privacidad de los datos. La red B de Alastria [20] basada en tecnología Hyperledger Besu [21] es una implementación de esta tipología.

La parte privada de la *blockchain* está formada por unos participantes en concreto, que se conocen entre ellos y tienen permiso de participación y de lectura de la ledger, mientras que en la parte pública se dan dos situaciones, o bien la parte pública solo tiene acceso de lectura o, en función del caso de uso, se encuentra abierta a la participación y lectura de cualquier individuo. La alianza Enterprise Ethereum Alliance [22] sigue este modelo de *blockchain*, combina la blockchain pública de Ethereum con las plataformas *blockchain* privadas de los integrantes de la alianza, como es el caso de BBVA.

Una implementación de red híbrida es la Sidechain. La desventaja que presentan las redes públicas es que en su consenso viene implícito el uso de criptomonedas. Los casos de uso que implican un elevado número de transacciones provocan que una red pública no sea viable económicamente. Por lo que se opta por realizar una gran parte de transacciones en una *blockchain* privada, y hacer transacciones de sincronización hacia la *blockchain* pública.

La categoría de **permisividad** se aplica principalmente en las redes de carácter privado debido a que todos los participantes se les otorga una identidad única y se les pueden aplicar roles para asignar diferentes privilegios dentro de la red. Los roles que predominan afectan directamente a la participación en el protocolo de consenso y a la lectura de las transacciones. Las redes públicas no se les aplica ningún tipo de restricción.

La arquitectura **descentralizada** de una red está directamente relacionada con la característica del privilegio de acceso a la red. La red pública, puesto que el acceso no es restringido y permite que todos los participantes mantengan una copia de la *ledger*, son las que se encuentran distribuidas por completo. Las redes híbridas y las privadas están parcialmente centralizadas o centralizadas, respectivamente, ya que predominan los consorcios entre empresas.

2.3.5. Mecanismos de consenso

La tecnología *blockchain* es un sistema distribuido que permite de forma descentralizada la transmisión de valor entre los nodos que forman parte de la red. El procesamiento y la gestión de las transacciones se encuentra descentralizada ya que no es necesario una entidad central que supervise la veracidad de las transacciones. Esto es debido a que los nodos participantes de la red son los encargados de alcanzar el consenso y asegurar que todas las copias del libro distribuido comparten el mismo estado.

La recopilación de requisitos y el problema que se quiere abordar utilizando la tecnología *blockchain* juegan un papel importante en el diseño. Durante el diseño de una aplicación *blockchain* se definen el conjunto de reglas y normas que los nodos deben cumplir para que una transacción sea válida. Este conjunto de normas se pre-establece al inicio de la blockchain siendo todos los nodos son conocedores de estas reglas.

Los algoritmos de consenso se encargan de asegurar que las reglas marcadas son respetadas, garantizando que todas las transacciones se realizan de forma fiable.
En otras palabras, las transacciones pueden ser verificadas y confirmadas en base a las normas que todos los participantes han acordado como válidas.

Por este motivo el consenso es la base de un sistema *blockchain* porque es el fundamento que permite que todos los participantes puedan confiar en la información que se encuentra grabada en él.

Los requisitos que deben cumplirse para proporcionar los resultados deseados en un mecanismo de consenso son los siguientes[23]:

- Acuerdo: Todos los nodos honestos deciden el mismo valor.
- Finalización: Todos los nodos honestos concluyen la ejecución del proceso de consenso y alcanzan una decisión.
- Validez: El valor acordado por todos los nodos honestos debe ser el mismo valor inicial que el propuesto por alemos un nodo honesto.
- Tolerante a fallos: el algoritmo de consenso debe tener la capacidad de ejecutarse en presencia de nodos que pierden la conexión a la red o funcionan de forma defectuosa. Por otra parte, debe tener la capacidad de efectuarse en presencia de nodos con intención maliciosa. (Problema de los generales bizantinos).
- Integridad: la decisión final tomada no puede modificarse ni tomarse más de una vez en un solo consenso.

Los mecanismos de consenso de los procesos distribuidos se agrupan en dos:

- La tolerancia a Faltas Bizantinas, más conocido en inglés como *Traditional Byzantine Fault Tolerance (BFT)-based*. Este método permite alcanzar el consenso cuando se presenta la situación de fallos o intento malicioso. Los participantes del sistema proponen sus valores, se comunican entre sí y acuerdan un valor de consenso único.
- Basado en la elección de líderes, en inglés *Leader election-based*. Los nodos compiten para ser escogidos mediante una lotería de elección, el nodo escogido propone el valor final de la transacción. Este método suele necesitar operaciones de cálculo intensivo y los nodos suelen ser recompensados por el esfuerzo de conmutación.

Actualmente, *blockchain* ha aplicado los mecanismos de consenso existentes para procesos distribuidos y ha implementado una amplia variedad de algoritmos según las necesidades. Los más implementados en la tecnología *blockchain* son los siguientes:

- *Proof of Work* (PoW):

Este protocolo utiliza la capacidad de computación de los nodos que participan en la red. Se propone un reto criptográfico a todos los nodos, el primero que consiga solucionar ese reto, agrega un bloque a la cadena con todas las transacciones pendientes. Este bloque es incorporado en la ledger, y es notificado y replicado a los demás nodos. Por el hecho de

haber generado un bloque y agregarlo a la red es recompensado, normalmente con criptomonedas. Por esta razón el protocolo se llama prueba de trabajo, porque los nodos compiten tratando de resolver un problema criptográfico a base de calcular y realizar muchos intentos por segundo. En este mecanismo no es necesario que exista confianza previa entre los nodos. El sistema presenta varias desventajas, la que más preocupa es el gasto de recursos energéticos. Gran cantidad de nodos intentarán resolver el reto criptográfico, pero solo uno de ellos resultará el que agregue el bloque a la red. A parte de esto, el bloque que se agrega necesita ser confirmado un número determinado de veces por otros nodos, solo de esta forma se da el bloque como válido.
La implementación más conocida que aplica este mecanismo de consenso es el Bitcoin [7].

- *Proof of Stake* (PoS):

Este mecanismo de consenso se basa en escoger el nodo que haya participado más veces en la red. Para la elección se tiene en cuenta el número de participaciones exitosas que ha realizado un nodo, o si se trata de una red con criptomonedas, se considera la cantidad de monedas y la antigüedad de estas. Este método aplica el razonamiento de que un nodo con gran volumen de participaciones satisfactorias tiene la intención de que la red funcione de forma correcta. Para ser candidato a crear un nuevo bloque, el nodo tiene que mostrar sus participaciones y en el caso de tener criptomonedas estas quedan congeladas durante un determinado tiempo después del consenso. De esta forma, no existe preferencia por los nodos que acaban de realizar propuestas o si el nodo ha intentado un ataque malicioso, pueden aplicarse sanciones. La implementación más conocida que emplea este mecanismo de consenso es el Peercoin [24].

- *Practical Byzantine Fault Tolerance* (PBFT)

El mecanismo PBFT [25] proporciona una solución al problema de los generales bizantinos. El objetivo es protegerse contra los posibles fallos del sistema mediante la toma de decisiones de forma colectiva. En la decisión se tiene en cuenta tanto a los nodos que proponen transacciones correctas, como los que pueden haber sufrido una desconexión del sistema o están tratando de hacer una transacción maliciosa. El hecho de que la participación sea en conjunto reduce la influencia de los nodos defectuosos. El consenso se logra cuando los nodos de la red que funcionan correctamente llegan a un acuerdo sobre los valores que cada uno ha propuesto.

Ha sido demostrado, que, para este algoritmo de consenso, el número de nodos necesario para que la red llegue a un consenso es el que cumple 3m+1. Siendo m el número máximo de nodos defectuosos. En otras palabras, esto significa que estrictamente más de dos tercios del número total de nodos de la red deberían ser honestos para llegar a un consenso.

El algoritmo de consenso PBFT organiza los nodos que participan en el consenso de forma secuencial. Si existen N nodos, se escoge el nodo principal (nodo líder) y se le atribuye el número 0, mientras que los demás N -1 nodos, llamados secundarios son ordenados indistintamente. Cualquier nodo puede ser elegido por el sistema como el principal. Los nodos líderes pueden ser sustituidos porque lo han sido durante un determinado tiempo o han realizado una cantidad de transmisiones de solicitud determinada. Si es necesario, la

mayoría de los nodos honestos pueden votar sobre la legitimidad del nodo principal actual y reemplazarlo.
Para mostrar de una forma más visual este protocolo se usa la figura 3. En la imagen se puede ver la existencia de 4 nodos: el principal es el Nodo 0, los secundarios son Nodo 1, Nodo 2 y Nodo 3.

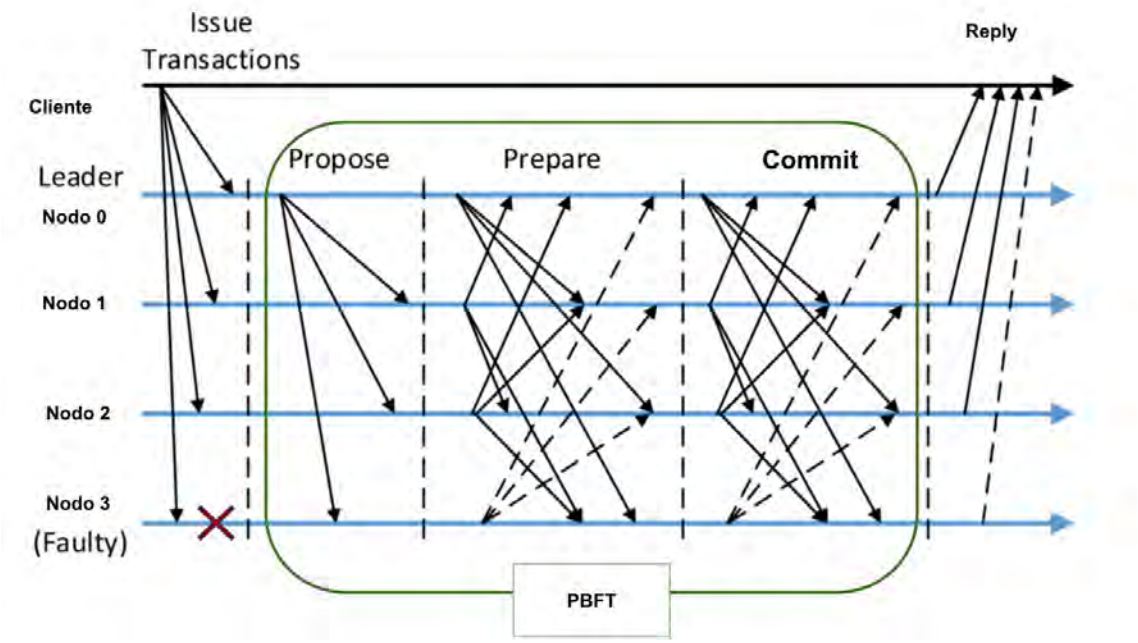


Figura 3. descripción del proceso *Practical Byzantine Fault Tolerance [25]*

El funcionamiento de este mecanismo de consenso se resume en cuatro pasos:

1. El cliente envía una solicitud de propuesta de transacción al nodo principal.
2. El nodo principal (0) transmite la solicitud a todos los nodos secundarios.
3. Cada uno de los nodos secundarios (1,2,3) ejecutan la solicitud y envían la respuesta al cliente.
4. La solicitud de propuesta de transacción se da como válida si el cliente recibe m+1 respuestas con el mismo resultado de diferentes nodos de la red. Considerando como m el número máximo de nodos defectuosos permitidos.

Durante el proceso anterior el algoritmo PBFT pasa por tres fases: *propose*, *prepare* y *commit*. [26]
En la fase de propuesta (*propose*) los nodos 1, 2 y 3 reciben la solicitud de transacción por parte del nodo 0. Todos los nodos secundarios evalúan la propuesta, la ejecutan y proponen una respuesta. En este ejemplo, el nodo 3 ha sufrido un fallo en su sistema y envía una respuesta errónea. Por lo que a la segunda fase llegan 3 respuestas correctas y una errónea.

La segunda fase *prepare* todos los nodos debaten las resoluciones de cada uno y llegan a un acuerdo común. Es decir, de las 4 respuestas, se escoge la más común y es enviada de nuevo a todos los nodos.

En la última fase commit, todos los nodos ejecutan individualmente de nuevo la propuesta de transacción y comparan con la respuesta escogida en la fase *prepare*. De esta forma, se garantiza que la primera respuesta escogida mediante consenso es la correcta y se asegura la fiabilidad del consenso. Aunque había un nodo que no ha funcionado correctamente, el consenso ha podido llevarse a cabo debido a que más de dos tercios del número total de nodos han llegado a la misma solución.

Por último, la respuesta se envía al cliente junto la comprobación de que se ha llegado al consenso por parte de los nodos.

Las ventajas que presenta este algoritmo frente los explicados anteriormente es la eficiencia energética. El PBFT logra el consenso distribuido sin la necesidad de realizar cálculos computacionales complejos ni un gasto energético desproporcionado. Adicionalmente, la resolución del consenso se lleva a cabo mediante la coordinación y el acuerdo entre todos los nodos. Por lo que no es necesario que el bloque generado sea revisado otra vez como es el caso de PoW.

2.3.6. Smart Contracts

Los *Smart Contracts* o contratos inteligentes son un programa informático totalmente descentralizado que facilita, asegura, hace cumplir y ejecuta un acuerdo entre dos o más partes. En el *smart contract* se pueden definir reglas y consecuencias como un documento legal tradicional, pero a diferencia de los contratos tradicionales, se ejecutan de manera automática y autónoma cuando se cumplen las reglas establecidas. Es decir, al cumplirse las normas, el *smart contract* pasa a ejecutarse de forma que todos los términos contractuales definidos en el mismo se llevan a cabo según lo definido y esperado, incluso en presencia de adversarios.
Los *smart contracts* reciben beneficios directos por el hecho de estar desplegados en una red distribuida DLT o en una *blockchain*. Entre las características principales encontramos:

- El programa informático no se encuentra controlado por ninguna de las partes que componen la red porque se encuentra descentralizado y distribuido.
- Son seguros e inmutables porque se almacenan de forma encriptada.
- La resolución del contrato es transparente, traceable e inmutable.

En resumen, los *smart contracts* reflejan las condiciones claras y las partes implicadas pueden consultar en todo momento el código y los resultados de la ejecución. Por como están programados se sabe que han sido ejecutados de forma autónoma y no han podido ser modificados. La aplicación de los smart contracts en ciertos procesos puede reducir de forma significativa los costes no solo económicos, sino también los temporales.

2.3.7. Implementaciones Tecnología Blockchain

Desde la primera implementación de blockchain, la tecnología ha avanzado a un ritmo muy rápido para tratar de cubrir aplicaciones que hasta hace uno o dos años eran impensables.

Para cubrir las necesidades requeridas de las posibles aplicaciones de la tecnología blockchain, existen en la actualidad una multitud de algoritmos de consenso:

- Ethereum: Es una plataforma abierta y descentralizada que sirve para manejar criptomonedas y desarrollar aplicaciones autorizadas. Se inició en 2015 y su criptomoneda es el Ether. El valor añadido que puedo aportar a Internet es que permite crear aplicaciones descentralizadas dApps que permiten a los usuarios relacionarse directamente entre ellos sin entidades centralizadas, como es google o facebook que gestionen el servicio.
- Quorum: es un proyecto de blockchain empresarial de Ethereum. Fue creada por la empresa financiera JP Morgan. Se caracteriza por ser customizable según el caso de uso que los participantes quieran darle a la red. Otro putno a destacar de este proyecto es que mantiene ocultos los detalles de las transacciones a nodos de la propia red. Su consenso está basado en el protocolo Raft como el proyecto HyperIdger Fabric.
- R3: consorcio consorcio que nació en 2014. Este consorcio creó el framework Corda. esta red está destinada a administrar contratos legales y transferir cualquier tipo de asset sin perder la privacidad de las transacciones.

2.3.8. Modelos de negocio de la Blockchain

Es habitual relacionar la tecnología con las criptomonedas o con el bitcoin, pero lo cierto es que existen otras muchas implementaciones que cubren aplicaciones como las que se muestran en este apartado, entre otras; identidad digital, digitalización de activos, sistemas de votación, gestión de la cadena de suministros, seguimiento de procesos de fabricación, gestión de datos, etc.

- **Seguimiento de activos físicos:**

La gestión de la cadena de suministros, en inglés *Supply Chain*, o en los mismos procesos de fabricación se utiliza la red *blockchain* junto con sensores para almacenar los procesos y las condiciones por los que ha pasado un *asset* durante su producción. Microsoft Azure presenta un servicio *cloud* de IoT junto con *blockchain* para este tipo de aplicaciones.

- **Procedencia de activos físicos:**

La procedencia y autenticidad de activos físicos puede ser comprobada mediante los sellos de autenticidad de las marcas o entidades certificadoras. Algunas veces estos productos son falsificados y se venden como auténticos. Para evitar el fraude, la entidad certificadora o la empresa que produce el elemento tiene un registro en la *blockchain* de todos los productos. La *blockchain* permitiría demostrar a los compradores que el producto es genuino. Esta aplicación es muy útil para el mercado de segunda mano.

- **Gestión de documentos y datos:**

La tecnología *blockchain* se está posicionando como la solución óptima para la gestión de documentos y de datos entre empresas y usuarios finales. Más concretamente existen sectores, como el sector de la salud, que necesitan un sistema para sustentar la fiabilidad, seguridad y transparencia de la información. Actualmente, el sistema sanitario es un sistema descentralizado y el historial clínico de cada paciente se encuentra distribuido por múltiples centros sanitarios. Esta situación trae consecuencias como repetir tratamientos que no han surtido efecto y repetición de pruebas que todo ello supone un elevado sobrecoste a la sanidad. Por otro lado, se puede dar la situación de necesitar asistencia médica en otra ciudad de un mismo país y no disponer del historial clínico del paciente.

Con un sistema basado en *blockchain* toda la red de centros hospitalarios tendrán acceso al historial de los pacientes. Ahora los pacientes son los que tendrán el control sobre sus datos siendo ellos mismos los que gestionan y controlan el acceso de sus historias clínicas. A su vez, se tendrá un registro de los medicamentos y tratamientos que han surgido efecto a cada individuo.

- **Registro de Activos digitales:**

La *blockchain* facilita el registro de los bienes materiales de una persona tales como el *real state*, el arte, joyas y otros objetos sujetos a un valor fiscal. Esta aplicación puede ser favorable para empresas aseguradoras y para facilitar el cobro de impuestos por parte del gobierno.

- **Transacciones económicas (Sector financiero)**

La aplicación de esta tecnología en las finanzas permite tener redes comerciales más abiertas, con la garantía de reducir costes y hacer los procesos más eficientes, se podrán dar servicios customizados en función de cada cliente.

2.4. Hyperledger

Hyperledger es un proyecto de código abierto de colaboración global creado con el propósito de fomentar el desarrollo y el uso de *blockchain* en diferentes sectores de la industria. Esta iniciativa internacional fue iniciada por Digital Asset e IBM, que actualmente está alojado por The Linux Fundation. El consorcio colaborativo reúne a empresas líderes en finanzas, banca, Internet de las cosas, cadenas de suministro, fabricación y tecnología.

Entre los diferentes proyectos que presenta Hyperledger, Fabric fue el primero de ellos en alcanzar la etapa activa en marzo de 2017.

2.4.1 Hyperledger Fabric

Hyperledger Fabric [19] es uno de los proyectos más conocidos del Proyecto Hyperledger. Es un *framework* de código abierto pensado para el desarrollo de soluciones empresariales que implementan la tecnología *Blockchain*. Desarrollada por 157 personas pertenecientes a 28 compañías diferentes. Entre ellas cabe destacar a los creadores originales IBM y Digital Asset, como resultado de una Hackathon.
Este *framework* ha sido usado para más de 400 prototipos y pruebas de concepto, y se han llevado muy pocos a producción en diferentes industrias.
Al igual que otras tecnologías *blockchain*, tiene una *ledger*, utiliza *smart contracts* y su sistema permite a los participantes administrar sus propias transacciones de assets.
Es de las pocas redes *blockchain* que presenta arquitectura modular, es decir, permite la customización de todos sus componentes. Desde el consenso hasta los nodos participantes. Esto también se traduce, en la escalabilidad, resiliencia y flexibilidad de la red según el caso de uso que se aplique, el número de participantes y los roles que desempeñan.
Entre sus características principales se encuentra la Privacidad. Los canales privados permiten a un grupo de participantes de la red crear *ledgers* distintos de transacciones que se pueden usar para proporcionar privacidad y confidencialidad de las transacciones. Esta es una opción especialmente importante para redes donde coexisten participantes competidores y no desean que todas las todas las transacciones se conozcan en toda la red. Por lo que una manera de privatizar la *ledger* entre participantes es crear un canal entre ellos: Todos los datos, incluida la información de transacción, miembro y canal, en un canal son invisibles e inaccesibles para cualquier miembro de la red que no tenga acceso explícito a ese canal.

El proyecto ha sido desarrollado en el lenguaje Golang y es entregado en formato de imágenes Docker para ser desplegado en un entorno Cloud.

2.4.1.1. Componentes específicos de Hyperledger Fabric

El *framework* Hyperledger Fabric presenta unos componentes específicos de su código respecto a otras *blockchain*. Estos han sido agrupados en:

- **Ledger**

La *ledger* de Hyperledger Fabric contiene el estdo de la red. Cada participante tiene una copia de esta. La *ledger* de Hyperledger Fabric tiene dos componentes bien diferenciados: el world estate y el *transaccion log*.

- El *world state* describe el estado del libro de registros en un momento específico de tiempo. Este componente es considerado como la base de datos.
- El *transaction log* se considera el historial de actualizaciones del *world state.* Este almacena las transacciones que ocurren anterior y posteriormente, poniendo en contexto el valor del *world state*.

En conclusión, la *ledger* es una combinación de las base de datos del *world state* y el historial del *transaction log*.

La *ledger* puede ser mantenida por un componente de tipo base de datos. LevelDB si se trata de entornos de test o pruebas de concepto, CouchDB si la red está pensada para entornos productivos.

- **Canales**

Los canales privados permiten crear redes blockchain con *ledgers* distintas. Cada canal tiene un *chaincode* distinto por lo que las transacciones son diferentes. El uso de esta característica proporciona a la red privacidad y confidencialidad en las transacciones.

Los peers se comunican y sincronizan el estado de la *ledger* a través de los canales de los cuales forman parte.

- **Nodes**

Los nodos son agrupaciones de infraestructura de la red Hyperledger. Esta agrupación se realiza por infraestructura "*trust domains*".

Existen tres tipos de nodos:

- Cliente: envía una propuesta de transacción (invocación) a los *peers endorsers* y también hace un broadcast de la propuesta de transacción al orderer.
- Peer: realiza transacciones, mantiene el estado y una copia de la *ledger*. Los *peers* reciben actualizaciones del estado por parte de los *orderers* en forma de bloques.
- Servicio de ordering o orderer: es el nodo que juega un papel importante en el flujo de las transacciones.

- **Peer**

Los *peers* son un elemento fundamental de la red Hyperledger fabric porque mantienen una copia de la *ledger* y ejecutan los *chaincodes*, los *smart contracts* de Fabric. Los *peers* se encargan de recoger los datos inmutables generados por los *chaincodes* y almacenarlos en la *ledger*. Cabe destacar que una característica particular de los *peers* es que son capaces de gestionar varias *ledgers* simultáneamente. Esto ocurre cuando un *peer* de una organización pertenece a más de un canal. Por esa razón, los peers pueden gestionar múltiples *chaincodes* simultáneamente.

Hay dos tipos de *peer*:

Los *endorsing peers* son aquellos capaces de ejecutar el *chaincode*, y de devolver a la red una propuesta de transacción. Mantienen la *ledger* y el *chaincode*.

Los *committing peers* solo contienen *ledgers* y reciben datos de otros *peer* que hacen modificaciones del *ledger* mediante *gossip protocol* o reciben los datos del servicio de *ordering*. El *gossip protocol* es el protocolo de comunicación que establecen los *peers* entre ellos para hacer comprobaciones de la *ledger*. Los *peers* están continuamente verificando

si tienen el último estado de la *ledger*, por eso lanzan *queries* continuas a la red que son respondidas por los otros *peers*.

Los *peers* se comunican con el cliente a través de SDK que puede ser empaquetado en las aplicaciones para poder interactuar con los *ledgers* y *chaincodes*.

- **Client**

Aplicación cliente propone transacciones a la red. Hace uso del SDK de Fabric para comunicarse con la red. El cliente se comunica con el SDK para leer o escribir datos en la red *blockchain* y en la *ledger*. El cliente presenta un certificado CA que avala que es un cliente válido que puede realizar transacciones en la *blockchain*.

El cliente suele implementarse en formato API.

- **Orderer**

En una red *Blockchain*, las transacciones tienen que ser escritas en la *ledger* compartida siguiendo un orden. El orden de las transacciones tiene que ser establecido para asegurar que las actualizaciones del World State son válidas cuando son entregadas a la red.

A diferencia de otras *blockchains* distribuidas, como Ethereum y Bitcoin, el consenso en Hyperledger Fabric se basa en algoritmos deterministas, cualquier bloque que sea generado por el *orderer* y validado por los peers tiene la intención de ser el bloque final y correcto que será escrito en la *ledger*. La *Ledger* en Hyperledger Fabric no se puede dividir, así como lo hacen otras *blockchains* como Ethereum.

El hecho de que el *endorsement* o propuesta de transacción se realice en los *peers* y no en el *orderer*, da una serie de ventajas a Fabric como la escalabilidad y elimina cuellos de botella que pueden suceder cuando la ejecución y el servicio de *ordering* se realizan en el mismo nodo.

Los tipos de *ordering service* son Solo, raft y Kafka.

- Solo presenta un único nodo *orderer*, esto provoca que esta implementación no sea tolerante a fallo. Por esta razón se considera como una implementación no apta para producción, pero es sin duda, muy útil para testear aplicaciones y Smart Contracts. Sobre todo, para crear Proofs of Concept es perfecto, pero si la intención es extender esta PoC a producción, es necesario implementar un nodo cluster de Raft.

- Raft es una implementación tolerante a fallos CFT (*Crash Default Tolerant*). Raft sigue un modelo de "leader and follower", donde se elige un nodo líder por canal y sus decisiones son replicadas por los seguidores. Raft introduce mejoras respecto Kafka ya que su diseño permite a diferentes organizaciones contribuir con sus nodos a un mismo servicio de *ordering*. Se implementó por primera vez en la versión de Fabric 1.4.1

- Kafka es muy parecido a Raft, es también CFT y es una implementación que utiliza la configuración de nodos "leader-follower". Kafka necesita ser un cluster para llegar a ser funcional. El cluster de Kafka se gestiona con la herramienta ZooKepper, este se asegura de que todos los kafkas tienen el mismo estado.

- **Certificate Authority (CA)**

La Certifcate Authoriry aporta a los participantes las credenciales necesarias para ser representados en la red como tal. La CA proporciona el *rootCertificate* y el *enrollment Certificate*.

- **Chaincode**:

Los contratos inteligentes de Hyperledger Fabric se llaman *chaincode*. Chaincode es un software que define los activos y sus transacciones relacionadas; en otras palabras, contiene la lógica empresarial del sistema. El chaincode tiene dos acciones muy marcadas. La primera el chaincode se instancia en el canal cuando todos los participantes lo aceptan como lógica de negocio de la red. La segunda, el *chaincode* se invoca cuando una aplicación tiene que interactuar con el libro contable. El *chaincode* se ha escrito en Golang en este proyecto, pero existe la posibilidad de hacerlo en NodeJS.

- **Consenso**

El consenso en la red Hyperldger Fabric está basado en el mecanismo de consenso PBFT (Practical Byzantine Fault Tolerance). Proporciona un mecanismo para que las réplicas de archivos se comuniquen , proporcionando a la red que se mantenga una copia coherente de la ledger, incluso en caso de corrupción. Como todos los componentes de la red Hyperldger Fabric es customizable y en el momento del diseño de la red Hyperledger permite a los participantes de la red elegir el mecanismo de consenso que mejor represente las relaciones que existen entre ellos.

El consenso está ideado para garantizar la concurrencia y el paralelismo en las transacciones. Presenta la posibilidad de que cada nodo ejecute transacciones antes de ordenarlas, esto permite a los nodos procesar múltiples transacciones simultáneamente. Esta ejecución concurrente aumenta la eficiencia de procesamiento en cada nodo y acelera la entrega de transacciones al servicio de ordenamiento. El consenso permite hasta un límite de procesamiento de 3.000 transacciones por segundo.

- **Membership Service Provider (MSP)**

Membership Service Provider (MSP) es un componente de la red que permite la validación y autenticación de las entidades y el acceso a la red. El MSP gestiona las ID de usuario y autentica a los clientes que desean unirse a la red. Esto incluye proporcionar credenciales a los clientes que propongan transacciones. El MSP hace uso de la Certificate Authority, la cual es una interfaz que verifica y revoca los certificados de usuario sobre una identidad previamente confirmada.

La interfaz predeterminada utilizada para el MSP es la API de Fabric-CA.

Hay dos tipos de MSP:

- Local MSP: Define usuarios (clientes) y nodos (*peers,orderers*). Define quién tiene poderes administrativos o de participación y a qué nivel.
- Channel MSP: Define poderes administrativos o de participación a nivel de canal.

2.4.1.2. Funcionamiento y flujo de las transacciones en Hyperledger Fabric

Hasta el momento se ha planteado el marco teórico de las redes *blockchain*, sus características y sus componentes. En este apartado se quiere platear un ejemplo que plasme el funcionamiento de una red *blockchain* privada implementada mediante tecnología Hyperledger Fabric.

El flujo de las transacciones que se ejecutan en Hyperledger Fabric consta de varios pasos. Para este caso de uso, se va a plantear la situación donde el participante A y el participante B se intercambian entre ellos manzanas.

Para definir la red *blockchain* cada cliente tiene un peer, el cual realiza transacciones e interactúa con la *ledger*, almacenando esas transacciones. Por otro lado, se establece un canal entre los dos *peers* para realizar transacciones entre ellos. Para realizar transacciones en la red, es decir, comunicarse con los *peers*, es necesario tener una API o cliente. Cada participante elabora una API y la registra mediante las *CA's* para poder comunicarse con los peer y hacer transacciones.

La lógica de negocio del mercado de las manzanas se documenta en formato *chaincode*, se instala en los *peers* y se instancia en el canal. Por lo que en el *chaincode* se define la lógica que define un conjunto de instrucciones de transacción y el precio acordado para una manzana.

Se ha definido una endorsement *policy* para este *chaincode*, siendo necesarios el endorsement de ambas organizaciones mediante los respectivos peerA y peerB.

Se inicia el proceso en la red *blockchain* cuando el participante A quiere comprar manzanas que están en la propiedad del participante B.

Inicio de la Transacción

1. El participante A hace una petición para comprar manzanas.

2. Se invoca la transacción mediante una aplicación cliente o API que interactúa con la red.

3. La petición apunta a ambos *peers*, ya que la transacción involucra a las dos partes que están en el canal.

4. La *endorsement policy* establece que ambos *peers* deben hacer el *endorsement* de cualquier transacción, por lo tanto, la solicitud se dirige a peerA y peerB.

5. Se construye la propuesta de transacción. La propuesta es una petición para invocar una función del *chaincode* con ciertos parámetros con la intención de leer o escribir en la *ledger*.

6. La aplicación cliente API empaqueta la propuesta con las credenciales del usuario para generar una firma única para esta propuesta de transacción

Ejecución de Transacción

1. Los *peers endorsers* que reciben la petición y realizan cuatro acciones: verifican que la transacción esté bien formada, que no se haya propuesto anteriormente (protección de ataque de *REPLAY*), que la signatura sea válida utilizando MSP y, por último, quien proponga la transacción esté autorizado a proponer tal transacción en el canal.

2. Los *peers* recogen los argumentos de la propuesta de transacción como argumentos de la función *chaincode* invocado.

3. El *chaincode* es ejecutado en el *world state* de la *ledge*r de cada peer para producir resultados de transacción que incluyen un valor de respuesta, un *read set* y un *write set*.

4.Hasta aquí, aun no se ha modificado la *ledger*, solo se **simula** el resultado para crear una propuesta de respuesta.

5.Los resultados de las transacciones junto con la firma de los *peers* que los generan se envían de vuelta a la aplicación.

Revisión de las propuestas

1.La aplicación verifica las firmas de los *peers* y compara las respuestas para determinar si son iguales.

2.Si el *chaincode* solo ha consultado la *ledger*, la aplicación inspecciona la respuesta, pero no emitirá la transacción a ser ordenada por el servicio de *ordering*.

3.Si el *chaincode* produce una escritura en la *ledger*, entonces la transacción tendrá que ser enviada al servicio de *ordering* para su ordenamiento.

4.Si la aplicación intenta emitir una transacción al servicio de *ordering*, la aplicación siempre determinará que dicha transacción ha cumplido con la *endorsement policy* antes de enviarla al servicio de *ordering*.

Creación de la transacción a partir de las propuestas.

1. La aplicación hace broadcast de la propuesta de transacción y de la respuesta dentro de un "mensaje de transacción" al servicio de *ordering*.

2.La transacción contiene los *read/write sets*, las firmas de los *peers* que firman la transacción y el ID del canal en donde se ejecuta la transacción.

3.El servicio de *ordering* no necesita revisar el contenido entero de la transacción para realizar su función, simplemente recibe transacciones desde distintos canales de la red, los ordena cronológicamente por canal y crea bloques de transacciones por canal.

Validación de la transacción

1.Los bloques de transacciones se envían a todos los *peers* en el canal.

2.Las transacciones dentro del bloque se validan para garantizar que se cumpla la *endorsement pol*icy y para garantizar que no se hayan producido cambios en el estado de la ledger para las variables del *read set* desde que la ejecución de la transacción generó el conjunto *read set*. Este paso se hace para garantizar que no haya lecturas fantasmas de valores que hayan podido ser modificados mientras se ejecutan las transacciones.

3. Las transacciones en el bloque son marcadas como válidas o inválidas

Actualización de la Ledger

1. Cada peer añade el bloque a la cadena del canal y, para cada transacción válida, los conjuntos de *write sets* se confirman en el *world state* de las bases de datos.

2. Se emite un evento para notificar a la aplicación cliente que la transacción (invocación) se ha agregado de manera inmutable a la cadena, así como una notificación de si la transacción ha sido validada o no.

Capítulo 3: Proyecto optimización de un proceso comercial

En el presente capítulo se detalla la gestión del proyecto basado en tecnología blockchain. Se presenta el alcance, los requisitos y especificaciones junto con el plan de trabajo que se ha llevado a cabo. También se detallan los recursos necesarios para el proyecto y las desviaciones ocurridas del plan inicial.

3.1. Alcance del Proyecto

El presente proyecto tiene la finalidad de comprobar la utilidad que una *blockchain* puede ofrecer y demostrar su aplicación a un caso real. En este sentido, el **presente proyecto se plantea en dos fases**, al inicio como una PoC (*Proof of Concept*) y luego como un proyecto en formato CaaS para ser desplegado en un entorno productivo. Debido a ello, el alcance requerido es ofrecer una implementación funcional del producto final, pero con la existencia de mejoras pendientes que serán detalladas más tarde en este documento.

El resultado final deseado, es una aplicación *back-end* basada en tecnología *blockchain* desplegada en el entorno Cloud Productivo y una aplicación web *frond-end* donde el cliente final pueda acceder, crear y gestionar las adendas comerciales. Desde esta misma aplicación el usuario debe ser capaz de modificar datos, hacer cambiar de estado la adenda, etc. Lo innovador de esta aplicación es que el almacenamiento y manejo de datos, así como la lógica de negocio, está controlada por una *blockchain*.

3.2. Definición del Proyecto

El objetivo de este proyecto es desarrollar un sistema que optimice un contrato comercial entre dos filiales de Caixabank mediante el desarrollo de una aplicación basada en tecnología *Blockchain* en un entorno Cloud productivo. Para ello, en el siguiente punto se especifican ciertos detalles de la estructura de la herramienta de adendas que se quiere implementar.

Cabe destacar, que la herramienta final se compone de dos partes bien diferenciadas: Front-End y Back-End. En el Front-End se encuentra todo el desarrollo relativo a la aplicación web: diseño de la interfaz, control de usuarios y roles, control de acceso, recopilación de datos, manejo de errores de usuario, etc. En el Back-End se encuentra todo el desarrollo relativo a la *blockchain*, que es el punto innovador de este proyecto. Las tareas relacionadas con este apartado son: el desarrollo de la infraestructura de Hyperledger Fabric, permitir la escritura y la lectura de datos de la *blockchain*, así como programar la lógica de las adendas (*Smart Contract*), y proveer el acceso a la *blockchain* para que otras aplicaciones puedan utilizarla (API).

Es necesario mencionar que en este documento se va exponer solamente detalles sobre la implementación del Back-End. Esto es por dos motivos. En primer lugar, porque el método de implementación del Front-End ha seguido todos los procedimientos y tecnologías estándar de ITNow. En segundo lugar, la implementación del Front-End ha sido realizada por un proveedor externo, mientras que el Back-End se ha llevado a cabo en ITNow por el departamento de innovación. Por lo tanto, no se tiene conocimiento de los detalles de su implementación.

Para finalizar, el proyecto se puede dividir en dos grandes fases. La primera fase es la de familiarización con la tecnología blockchain y tecnología cloud, donde se lleva a cabo la implementación y el desarrollo de una primera versión del *back-end* en un entorno de pruebas. Es decir, el proyecto en esta fase se encuentra en formato PoC. Una vez finalizada la aplicación, el departamento de mejora continua de ITnow pudo demostrar la optimización en tiempo y en gestión de los contratos, por lo que se dió luz verde a empezar la fase dos de despliegue en producción.

La segunda fase del proyecto es la que engloba todos los procesos y cambios de estructura, para adaptar el código de Hyperledger Fábric y los componentes de la aplicación, con el objetivo de despliegue en el entorno productivo. Al tratarse de la primera aplicación con esta tecnología tan novedosa, la empresa no tenía un código de buenas prácticas para el despliegue en el entorno *on-premise.* Por lo que gran parte del código de la fase 1 tuvo que pasar varias inspecciones y auditorías de seguridad. Esto originó varios cambios que serán explicados en el apartado de desviaciones del plan inicial.

3.3. Requerimientos y especificaciones

Los requerimientos y especificaciones del proyecto se definen en los siguientes puntos:

1. Implementación de la aplicación corporativa para llevar a cabo el proceso comercial de adendas mediante tecnología *blockchain*.
    a. El aumento de la trazabilidad del documento aportando a los usuarios de la aplicación el completo control de las versiones del mismo, así como de los archivos adjuntos.
    b. La optimización del tiempo debido al completo conocimiento del estado o fase en el que se encuentra el documento. Es decir, todos los participantes y personal involucrado en el proyecto tienen la misma versión del contrato comercial y de los archivos adjuntos.
    c. La optimización del tiempo debido a dos factores: al completo conocimiento del estado o fase en el que se encuentra el documento.
    d. Las dos premisas anteriores deben ser significativas para que el proyecto sea viable económicamente.
2. Completo conocimiento y adaptación del código de HF para el cumplimiento del código de buenas prácticas del cloud privado, así como el cumplimiento de seguridad marcado por Caixabank.
    a. Creación entorno test parecido al cloud productivo de Caixabank.
    b. Proporcionar un código viable para ICP de la API-Rest, la base de datos y

c. Adaptación de la versión 1.4 de Hyperledger Fabric para ser desplegada y funcional en el cloud privado.
d. Estudio, desarrollo y adaptación de la versión 2.0.0 y 2.0.1 de Hyperledger Fabric para ser desplegada y funcional en el cloud privado.
e. Gestión de despliegue y de operatividad de la aplicación *blockchain* en el entorno cloud privado.

3.4. Gestión del Proyecto

La gestión del proyecto se ha realizado mediante la metodología de gestión de trabajo Scrumban que sirve para el desarrollo ágil de software. Scrumban es una metodología híbrida entre Scrum y Kanban. Es la transición de Kanban a Scrum, los principios básicos y las prácticas de esta metodología no están totalmente definidos por lo que en este punto se va a describir cómo se lleva a cabo en ITnow. Especialmente en el departamento de Innovación.

El equipo Scrumban está formado por:

- Propietario del producto: también conocido como *Product Owner*, en este caso el director del área de innovación.
- *Scrum Master*: realiza las tareas de project manager, gestiona el proceso de trabajo. Su objetivo es que el equipo esté en continuo funcionamiento y no tenga ningún bloqueo ni impedimento. El puesto de *Scrum Master* lo suele ocupar uno de los desarrolladores del equipo, normalmente el que tiene más conocimientos sobre la tecnología y tiene una visión transversal de otros equipos de la empresa.
- *Stakeholders*: es el cliente final al que se le entrega la aplicación, en este caso los departamentos comercial y legal de las filiales de Caixabank.
- Equipo *Scrum*: el equipo de desarrolladores que implementan el proyecto.

Scrumban presenta *sprints* que son intervalos establecidos de tiempo donde los desarrolladores se comprometen con el cliente a presentar *software* viable para la aplicación o el proyecto en cuestión. En el departamento de innovación la duración de los *sprints* es de tres semanas. Las historias de usuario son las tareas donde se define una implementación de código que es necesaria para avanzar en el proyecto. El *Backlog* es la lista de historias de usuario y tareas pendientes que se realizarán en un futuro. Durante cada *sprint* los desarrolladores tienen una lista de historias de usuario donde se detalla las tareas a cumplir. Al inicio del *sprint* el Propietario del Producto junto con el *Scrum Master* deciden qué historias de usuario se extraen del Backlog para que sean realizadas por parte del equipo *scrum.*

Con la finalidad de visualizar el estado del proyecto, se hace uso de un tablero Kanban. El tablero se encuentra dividido en columnas. Cada columna representa un paso en el flujo de trabajo y cada tarjeta representa una tarea. Según se muestra en la figura 4, hay cinco columnas:

- *Backlog* es la lista de tareas o de historias de usuario pendientes que no entran en el *sprint*. Son tareas que se tendrán que hacer en un futuro, donde tanto el

propietario del producto como el *scrum master* tienen que preparar y gestionar para que en el siguiente sprint no exista ningún impedimento.

- *To Do* es la lista de tareas o historias de usuario que entra en el sprint. No están asignadas a ningún miembro del equipo. Al finalizar el sprint deben estar completadas.
- *Doing* es la lista de tareas asignadas para el presente *sprint.*
- *Blocked,* todas las tareas que no pueden ser realizadas por algún motivo. Los motivos pueden ser muy diversos: por un fallo de infraestructura, por una dependencia de otro departamento, etc.
- *Done* son las tareas o historias de usuario que han sido finalizadas y presentadas al cliente. Por otro lado, como se trata del departamento de innovación, en esta columna también se ponen las tareas que han sido descartadas debido a que se ha demostrado que no son viables.

Figura 4. Kanban del departamento de Innovación en el que se detallan las tareas de los diferentes proyectos que se llevan a cabo durante un *sprint.*

La metodología que sigue es de cuatro reuniones principales:

- Reunión de Planificación de *Sprint*. Esta reunión sirve para añadir tareas al *backlog*. El objetivo de esta reunión es garantizar que las historias de usuario las puede ejecutar el equipo una vez entren en un *sprint.*
- Reunión diaria o "*Daily".* Como su nombre indica es una reunión diaria que tiene de duración de 15 a 20 minutos donde cada uno de los integrantes del equipo contestan las siguientes tres preguntas: ¿Qué hice ayer? ¿Qué haré hoy? ¿Qué impedimentos, bloqueos u obstáculos tienen mis tareas hoy?
  Las respuestas a estas preguntas sirven al *scrum master* y al propietario del producto para tener una visión del estado de las tareas y el estado del *sprint.*
- Reunión revisión de *Sprint*. La reunión se realiza al finalizar el *sprint,* las tareas en la columna *Done* se presentan primero al Propietario del Producto y se establece si

se cumplen los requisitos. Si se cumplen los requisitos, el código se entrega a los *Stakeholders* y las tareas pasan a estar finalizadas, por lo tanto, se eliminan del proyecto. Por último, las tareas que se encuentran en el *backlog* pasan a la columna *To Do.*

- Reunión de Retrospectiva de *Sprint*. Son reuniones que se hacen unos días después de finalizar un *sprint*. El objetivo es reunir solo el equipo *scrum* junto con el *scrum master* para reflexionar sobre el *sprint* anterior y descubrir cómo mejorar el proceso de gestión. Se hace una lista de lo que ha ido bien, lo que ha ido mal y qué se puede mejorar. Permite al equipo expresarse sobre lo bueno o lo malo, ayuda a centrarse en su rendimiento e identificar formas de mejora contínua.

3.4.1. Tareas del Proyecto

Las tareas principales que se necesitan cumplir para lograr el alcance del proyecto son las siguientes:

La Fase 1 del proyecto es donde se realizan los primeros diseños e implementaciones de infraestructura en formato PoC. Una vez finalizado este proyecto se estudia la viabilidad de ser desplegado en producción.

- Definición de los requisitos del proyecto. Esta tarea se lleva a cabo mediante reuniones con el cliente final de la aplicación, los *Stakeholders* del proyecto. Durante las reuniones se conoce la problemática actual que presenta el flujo comercial, las necesidades y mejoras necesarias, además de definir el alcance del proyecto y reunir las especificaciones del producto final.
- Planificación. Es de vital importancia saber cómo se va a organizar el proyecto, el tipo de gestión de proyectos que se va implantar y las reuniones de seguimiento por parte del cliente.
- Petición de recursos. Para realizar la PoC de este proyecto se ha necesitado solicitar recursos internos de la empresa en forma de máquinas virtuales.
- Preparación entorno *Cloud test*. Es necesario preparar y configurar el entorno proporcionado por la empresa. Las máquinas virtuales son proporcionadas con lo mínimo, por lo que esta tarea conlleva el despliegue de todos los servicios necesarios para crear un entorno parecido al cloud corporativo.
- Estudio de las tecnologías. La tarea engloba el estudio exhaustivo de todas las tecnologías, para entender su funcionamiento y escoger en consecuencia las implementaciones más convenientes para el proyecto.
- Definición, diseño e implementación de la infraestructura *blockchain*. Se diseña la red *blockchain* con los componentes necesarios para el caso de uso y la implementación en formato *cloud*.
- Diseño de la lógica de negocio. Se reúnen las características esenciales del documento comercial para definir la lógica de negocio que se plasma en el *chaincode*. Esta tarea se realiza conjuntamente con los *Stakeholders* y el departamento legal de las filiales.
- Definición, diseño e implementación de la base de datos.
- Definición, diseño e implementación de la *API-Rest*.

- Pruebas de infraestructura. Se realizan pruebas de estrés y de funcionamiento a los componentes de la aplicación por separado: red *Hyperledgerfabric*, *API-Rest* y base de datos.
- Pruebas globales del sistema. Se realizan pruebas de estrés y de funcionamiento a la aplicación en general con todos los sistemas funcionando. Estas pruebas las valida el cliente final.
- Conclusiones y resultados de la PoC. Para finalizar la fase 1 del proyecto se estudia.

La Fase 2 es la que estudia la migración al entorno *cloud* productivo y se realizan las tareas en consecuencia. Para ello es imprescindible llevar a cabo:

- Auditoría de Seguridad y validación de la aplicación por parte del departamento de infraestructura *cloud*.
- Validación del Diagrama topológico de la infraestructura.
- Requisitos de despliegue en el entorno *cloud* productivo.
- Petición de recursos en el *cloud* corporativo.
- Adaptación de los componentes de la aplicación según los requisitos de operación y gestión del cloud corporativo. HA, contingencia en caso de desastre, definición de SLAs del producto.
- Adaptación del código según los requisitos de seguridad y buenas prácticas corporativas.
- Diseño e implementación de la monitorización, definición de scripts de automatización de despliegue y gestión de posibles fallos de la aplicación.

3.4.2. Recursos

Los recursos necesarios para el desarrollo del proyecto se presentan diferenciados entre recursos materiales y recursos humanos.

Los **recursos materiales** del proyecto se pueden resumir en la siguiente tabla 1 :

Tabla 1. Recursos del proyecto

Infraestructura	Recursos
3 máquinas virtuales del domino de ITnow	8 vCPU 8 GB de RAM First Disk 100GB (SAN) Sistema Operativo: Red Hat Enterprise Linux v7.7
Network File System (NFS)	100 GB

Los **recursos humanos** que han sido necesarios para desarrollar el proyecto forman parte del departamento de innovación. El equipo está formado por tres integrantes los cuales han ocupado las siguientes posiciones en el proyecto:

Tabla 2. Recursos humanos del proyecto *blockchain*

Integrante departamento de innovación	**Posición Scrumban**	**Dedicación horas en el proyecto**	**Tareas realizadas e implicación en el proyecto**
Laura Esbri	*Scrum Master Developer*	50%	Desarrollo e implementación de la arquitectura cloud de *Hyperledger Fabric*. Gestión del desarrollo del *Chaincode*, de la *API-Rest* y base de datos. Gestión de tareas bloqueadas.
Integrante A	*Developer*	50%	Desarrollo e implementación de la API-Rest junto con el SDK de *Hyperledger Fabric*. Desarrollo e implementación de la base de datos.
Integrante B	*Developer*	25%	Desarrollo e implementación del *chaincode*.
Integrante C	*Product Owner*	5%	Gestión de las tareas del *Backlog.*

3.4.3. Estudio económico y coste en horas del TFM

A continuación, se detallan los costes totales del proyecto.

3.4.3.1. Coste de personal

El coste del personal se puede observar en la siguiente tabla 3.

Tabla 3. Estudio económico. Coste personal. El coste diario incluye cargas sociales.

		% Dedicación	Coste Anual	Coste Diario	# Días	Total Costes
Laura Esbri	*S M D*	50%	40.000,00 €	54,79 €	519	28.438,36 €
Integrante A	*Developer*	50%	30.000,00 €	41,10 €	519	21.328,77 €
Integrante B	*Developer*	25%	30.000,00 €	20,55 €	519	10.664,38 €
Integrante C	*Product Owner*	5%	120.000,00 €	16,44 €	519	8.531,51 €
						68.963,01 €

3.4.3.2. Coste del Hardware

El presupuesto del coste de hardware es orientativo debido a que el entorno de pruebas construido en el proyecto se ha realizado en las máquinas on-premise de la empresa ITnow.

Por lo tanto, no se puede ofrecer el coste real de las máquinas por razones de confidencialidad y por la complejidad del cálculo. Por ello, se ha visto conveniente escoger el precio orientativo de la cuenta de IBM Cloud corporativa ya que esta es utilizada en otras pruebas de concepto del departamento de innovación. El coste tiene en cuenta 3 máquinas virtuales y el NFS.

En el Anexo 1 se aporta la evidencia de coste de las máquinas del IBM Cloud.

Tabla 4. Estudio económico. Coste del Hardware.

			Fecha Inicio	*01-jul-19*	
			Fecha Fin	*01-dic-20*	
			Total días	*519*	
	Número Máquinas	**Coste / hora**	***Horas / día***	**# Días**	**Total Costes**
Renting Máquinas					
	3	0,33	24	519	12.331,44 €
NFS					
	1	0,0193	24	519	240,40 €
					12.571,84 €

3.4.4. Planificación

Planificación del proyecto en general se puede observar en la siguiente imagen:

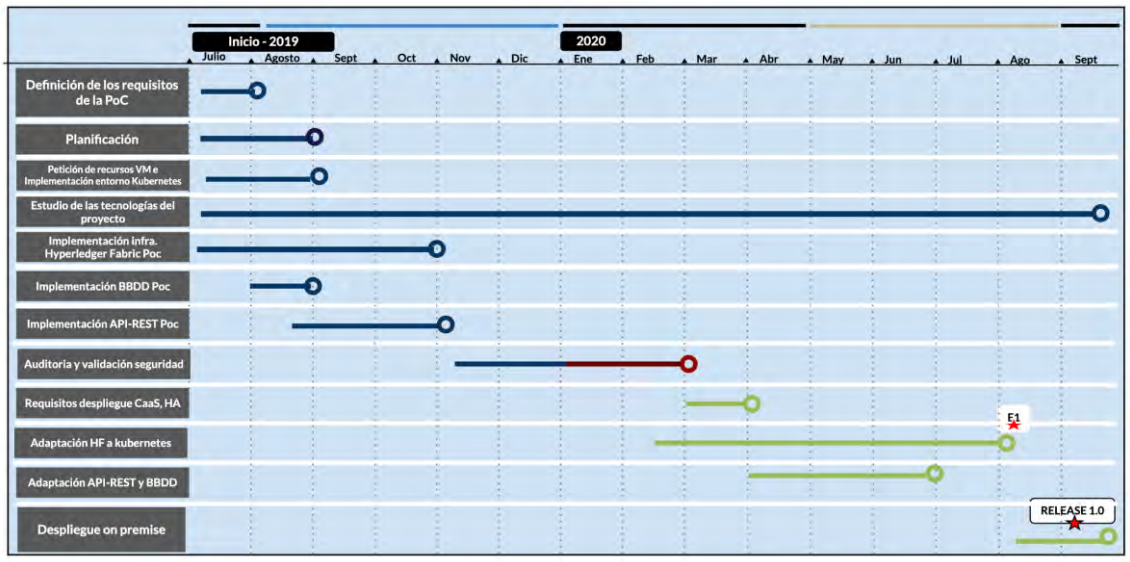


Figura 5. Planificación de las dos fases del proyecto

3.4.5. Desvío del Plan Inicial

El proyecto de la aplicación basada en *blockchain* para un proceso comercial entre dos filiales de *Caixabank* ha sufrido diversas afectaciones que han incidido en la planificación inicial. Es cierto que, en cualquier proyecto, y más especialmente en uno de innovación, existen riesgos que se materializan en modificaciones del plan de trabajo. En este proyecto, especialmente, se ha visto afectado por dos motivos:

- La falta de madurez del código *Hyperledger Fabric* v.1.4 para ser desplegado sobre el orquestador de contenedores Kubernetes.
- El incumplimiento de las normas de seguridad por parte de la implementación del código de *Hyperldger Fabric* v.1.4 en Kubernetes.

Estos dos impedimentos han significado meses de búsqueda de soluciones alternativas, sin encontrar un diseño e implementación que cumpliera el requisito de ser desplegado en un entorno de Kubernetes y, a su vez, cumpliera los requisitos de seguridad del entorno *Cloud* productivo de *Caixabank*.
La solución de los impedimentos mencionados se llevó a cabo debido al lanzamiento de la versión de *Hyperledger Fabric* 2.0.1 el día 26 de febrero de este año 2020. Este lanzamiento tenía la intención de solucionar los problemas de seguridad, pero el código estaba optimizado para el despliegue en contenedores Docker. En consecuencia, se tuvo que colaborar juntamente con los desarrolladores de *Hyperledger Fabric* mediante el chat de *Hyperledger [27]* para encontrar una solución viable para un entorno de Kubernetes. Gracias a esta colaboración [28], junto con un compañero de profesión que trabaja en *Allianz Technology* en Múnich, se pudo implementar una versión de código de *Hyperledger Fabric* perfectamente optimizada para un entorno de Kubernetes.

Capítulo 4: Desarrollo de la solución

En el presente capítulo se detalla el diseño, desarrollo e implementación de la aplicación *blockchain*. Se explica el diseño y la arquitectura de la red *Hyperledger Fabric* planteada en la fase inicial del proyecto en formato PoC, junto con las soluciones alternativas propuestas. Adicionalmente, se muestra el desarrollo y la implementación de la segunda fase del proyecto con el código productivo. Por otro lado, se expone el desarrollo de la API-Rest y de la base de Datos.

4.1. Objetivo de la prueba de Concepto

El objetivo de la **primera fase del proyecto** es desarrollar una prueba de concepto sobre una aplicación que permite la gestión de un proceso comercial entre dos departamentos de empresas diferentes de *Caixabank*. Esta aplicación se puede dividir a grandes rasgos en frontal web y *back-end*. El *front-end* ha sido implementado por una empresa externa a *ITnow*, dedicada a diseñar herramientas corporativas de la Caixa. El *back-end* ha sido implementado en su totalidad por el departamento de innovación de *ITnow*.

El proceso comercial identificado que se quiere implementar mediante tecnología *blockchain* consta de dos partes. La primera hace referencia a un documento que se adjunta con el proceso comercial. Este documento adjunto presenta un formato parecido a una factura que suele tener detallado los elementos de compra y su importe.

La segunda hace referencia a la edición y redacción del documento, llamado adenda. Para el desarrollo de esta parte los dos departamentos en cuestión hacen uso de una plantilla base escrita en word por el departamento legal de *Caixabank*. El departamento A realiza la petición de inicio del proceso comercial mediante un correo electrónico al departamento B, una vez se inicia el proceso comercial, el departamento B rellena a mano el documento, adjunta el documento tipo factura y lo envía por correo corporativo al buzón del departamento A. Seguidamente, el departamento A tiene que realizar una serie de comprobaciones y, si se da el caso, modificar el documento word. Una vez se han finalizado las modificaciones por parte del departamento A, se envía el documento modificado con el fichero adjunto.

Por la tipología de contrato, el documento word tiene que pasar de media 6 revisiones antes de enviar a firmar por *customer*. Por lo que muchas veces se pierde la trazabilidad y el estado del proceso, es decir, se da la situación que ninguno de los dos departamentos sabe la versión última del documento, así como, a veces se deja de adjuntar la factura.
Adicionalmente, este documento del tipo factura presenta una fecha de caducidad. Por esta razón, algunas veces no se han acabado de hacer las revisiones y la fecha de la factura ya no es válida.
Así mismo, para que el proceso se dé como finalizado completamente, se tiene que firmar físicamente por los directores financieros de ambas filiales, llamadas *customer* en este proceso.

El proyecto tiene como objetivo crear una herramienta basada en *blockchain* que centralice este proceso y elimine las ineficiencias de las comunicaciones por correo y las ediciones mediante Microsoft Word. La solución presenta la siguiente forma:

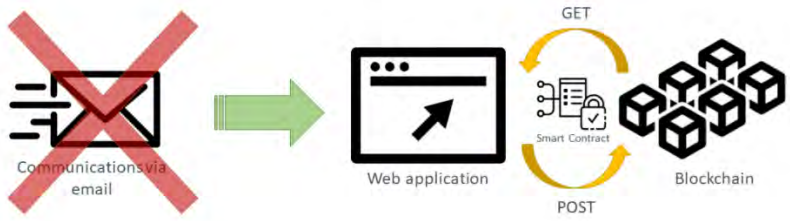


Figura 6. TO-BE de la solución técnica.

4.1.1. Flujo del proceso

Con la finalidad de que el lector pueda comprender el flujo del proceso comercial anteriormente explicado, se detalla mediante el flujograma de la Figura 7.

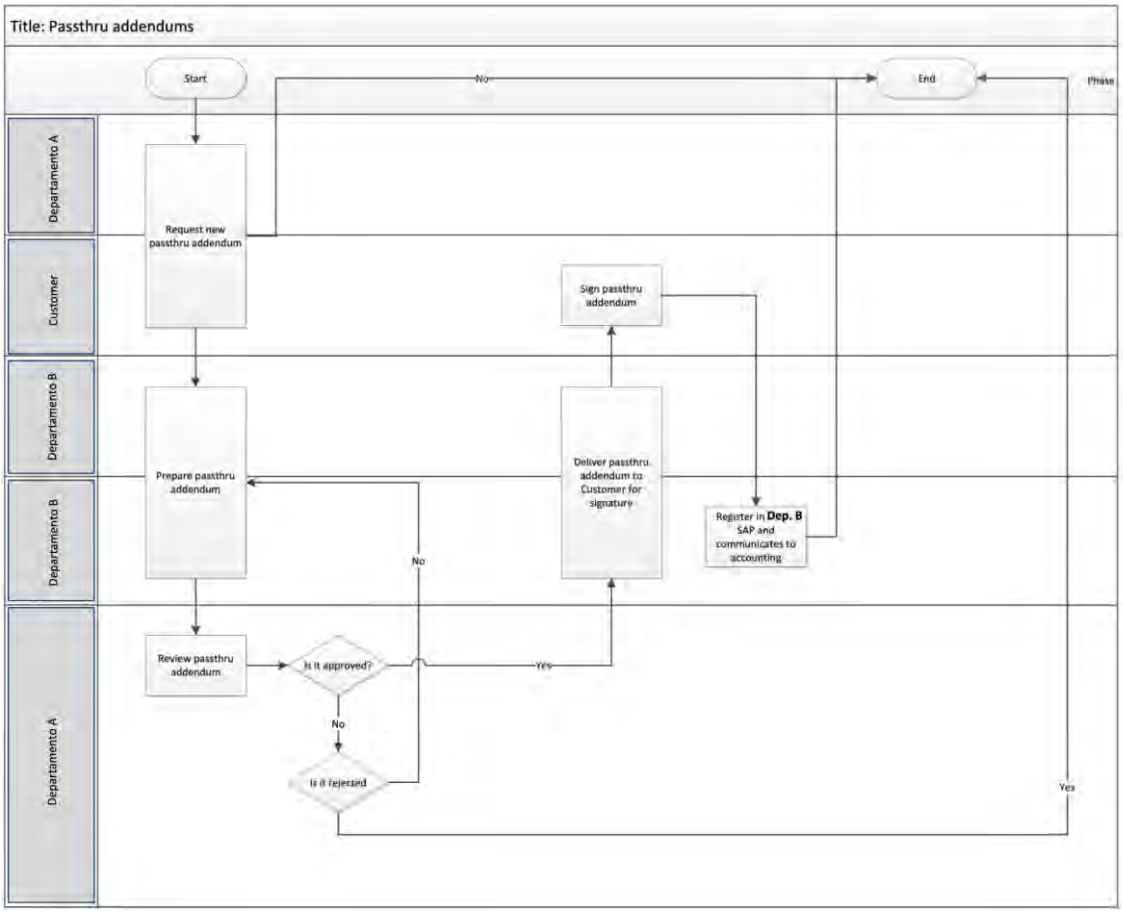


Figura 7. Flujograma del proceso de adendas comerciales.

4.2. Implementación del entorno de pruebas

La prueba de concepto se ha realizado sobre tres máquinas virtuales proporcionadas por la empresa. La configuración de las máquinas se ha llevado a cabo desde el departamento de innovación. Para realizar un entorno de pruebas lo más parecido al entorno *Cloud* corporativo se han realizado una serie de reuniones con el departamento de infraestructura de *ITnow* para concretar los requisitos de dicho entorno.

4.2.1. Preparación entorno Máquinas Virtuales

Antes de empezar con la instalación del entorno cloud hay que preparar el NFS. El *Network File System* es un protocolo que permite la conexión de varios servidores que pertenecen a la misma red para que puedan acceder a ficheros remotos como si se tratara de locales.

Una de las máquinas que se dipone tiene el recurso de 100Gb, por lo que esta es el servidor NFS y las otras dos restantes son el cliente NFS.
Se instala en las tres máquinas virtuales el paquete de herramientas nfs-utils mediante el siguiente comando:

```
yum install nfs-utils
```

El servidor NFS es el encargado de configurar quién tiene acceso al recurso. Esto se hace editando el fichero /etc/exports. Siguiendo la siguiente estructura:

```
directorio_nfs ip_cliente(opciones_cliente)
```

Se escoge el directorio /mnt de la raíz para ser compartido.

```
/mnt 10.1XX.XXX.XXX(rw,no_root_squash)
/mnt 10.1XX.XXX.YYY(rw,no_root_squash)
```

Las opciones_cliente seleccionadas son:

- rw: permiten los permisos de lectura y escritura.
- no_root_squash: permiten que los usuarios conectados puedan tener acceso root.

Cada vez que se actualice el fichero /etc/exports , se tendrá que ejecutar el comando siguiente en el terminal:

```
exports -r
```

Finalmente, para acabar con la instalación, desde las máquinas virtuales clientes NFS, se lanzan los siguientes comandos en cada una de ellas:

```
mount -t nfs -o rw,sync,hard,intr 10.1XX.XXX.XXX:/mnt /mnt
```

```
mount -t nfs -o rw,sync,hard,intr 10.1XX.XXX.YYY:/mnt /mnt
```

Se puede realizar un cat del fichero /etc/fstab en una de las máquinas y se puede comprobar cómo el comando `mount` anteriormente lanzado ha montado el directorio /mnt de la máquina servidor NFS.

```
cat /etc/fstab
```

```
10.1XX.XXX.YYY:/mnt   /mnt   nfs  rw,sync,hard,intr      0     0
```

4.2.2. Preparación entorno Cloud

La instalación del entorno *Cloud* se va a realizar instalando sobre el sistema operativo *Red Hat Enterprise Linux v7.7* el software *Docker* y el orquestador de contenedores Kubernetes.

4.2.2.1. Conceptos de Docker

Docker [29] es un *software* libre y de código abierto que permite crear aplicaciones y desplegarlas dentro de contenedores ligeros y portables. Estos contenedores pueden ejecutarse en cualquier máquina, independientemente del sistema operativo que tenga, facilitando el despliegue de varios tipos de aplicaciones que necesitan diferentes prestaciones en un mismo entorno. *Docker* se ejecuta por encima de los recursos más básicos de un servidor, el sistema operativo, mientras que los aspectos específicos de cada aplicación se encapsulan dentro del contenedor. Este *software* libre se originó debido al aumento de las aplicaciones en los centros de datos y las dependencias de estas.

Los contenedores se crean a partir de una imagen la cual lleva todos los requisitos necesarios para que el contenedor mantenga una funcionalidad determinada. Desde el sistema operativo hasta posibles módulos o paquetes. Existen imagenes base por defecto que se pueden encontrar en formato *Open Source* en *DockerHub*. El repositorio por excelencia de imágenes de contenedores *docker* es *Dockerhub*, múltiples empresas mantienen las imágenes *open source* en el repositorio.
Si es necesario y necesitamos crear una imagen customizada lo podemos hacer mediante el fichero *Dockerfile*. El fichero *Dockerfile* permite crear una nueva imagen personalizada a partir de una imagen base.

4.2.2.2. Instalación de *Docker*

La instalación de *Docker* se realiza con los siguientes comandos. Primero, se agrega el repositorio con el proyecto docker para el sistema operativo *Red Hat Enterprise Linux v7.7*. Este repositorio se trata del interno de la empresa, por lo que el comando no puede ser mostrado.

Seguidamente, se instala y se inicia la *Docker Engine* mediante los siguientes comandos:

```
yum install docker
systemctl start docker
```

Finalmente, se comprueba que la instalación ha ido correctamente lanzando el siguiente comando:

```
systemctl status docker
```

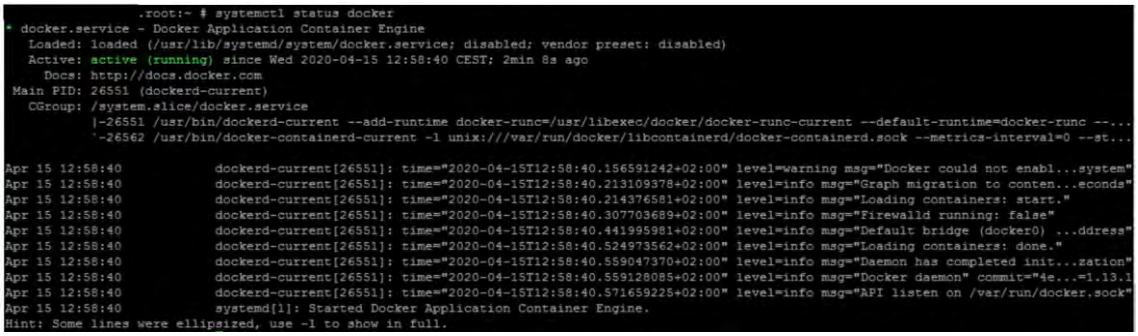


Figura 8. resultado del comando systemctl status docker

En la figura se puede ver que el sistema *Docker* se encuentra levantado y ejecutándose en la máquina.

Mediante el siguiente comando se puede ver la versión de *Docker* que ha sido instalada:

```
docker version
```

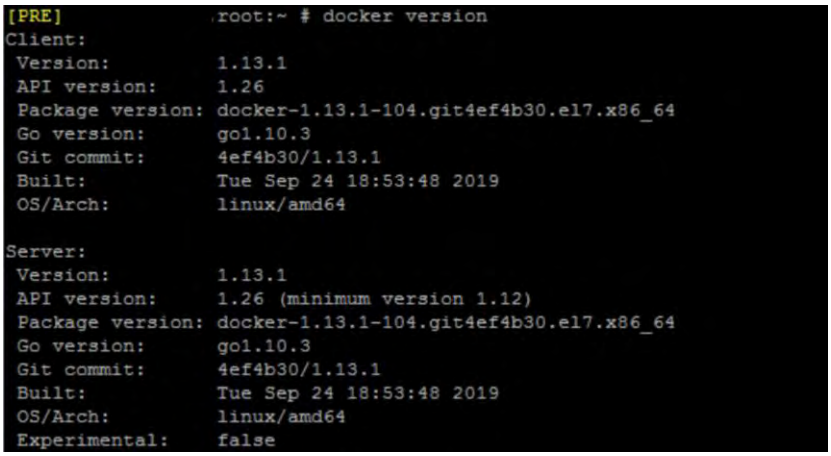


Figura 9. Docker version

4.2.2.3. Conceptos de Kubernetes

Kubernetes [29] es un sistema de código libre para la automatización del despliegue, el escaldado y gestión de aplicaciones en contenedores que se ejecutan sobre servidores. Funciona con una serie de tecnologías de contenedorización, pero la más utilizada es Docker. La también llamada plataforma de microservicios, gracias a su arquitectura, es capaz de ejecutar los sistemas distribuidos de forma resiliente, encargándose de los posibles fallos y volviendo a desplegar la aplicación. A parte, se encarga de gestionar los recursos de memoria, de cpu y de red que ofrecen los servidores donde corren.

La infraestructura que se va a desplegar en este proyecto se encuentra en formato objetos de Kubernetes. Por lo que es necesario situar al lector con los conceptos básicos y la nomenclatura específica de la tecnología. Los objetos que se van a describir permiten desplegar microservicios totalmente funcionales.

La arquitectura de Kubernetes plantea una estructura *master-slave*. De esta forma se generan los *cluster*. El entorno cloud que se implementa en este proyecto sigue la siguiente estructura:

- 1 Máquina Virtual como *master*
- 2 Máquinas Virtuales como *workers* o nodos.

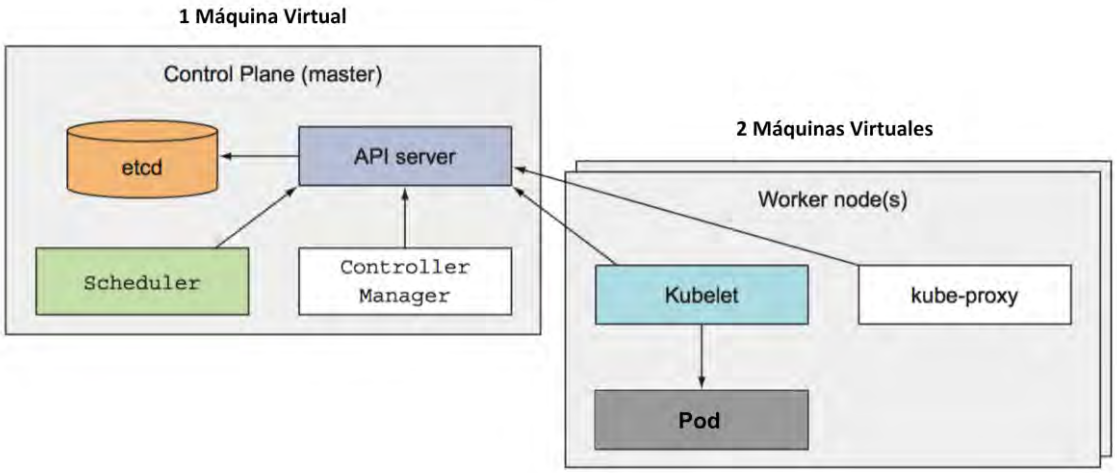


Figura 10. Arquitectura cluster

Para conseguir la gestión de los contenedores *docker*, Kubernetes presenta una serie de componentes que se encuentran instalados en el master o en los nodos dependiendo de su funcionalidad.

Los componentes que se despliegan en el máster son los siguientes:

- **etcd:** es la base de datos de tipo clave-valor del master. Se emplea para recoger y almacenar las configuraciones globales del cluster.
- ***API Server*:** es el componente central de kubernetes que procesa y valida las peticiones que son lanzadas desde los *workers* y desde la *command line interface*.
- **kube-scheduler:** se encarga de decidir donde se despliegan los contendores en función de los recursos de los *workers*.
- **kube-controller-manager:** mantiene un bucle de control que observa el estado en global del cluster a través de la *API Server*, si en necesario, realiza cambios para dejar el estado deseado del cluster.

Los componentes que se despliegan en los nodos son los siguientes:

- **kubelet:** es el sistema responsable de monitorear los estados de los pods y mantenerlos desplegados en el nodo en cuestión. Si el kubelet detecta la caída de un *pod*, su función es re-desplegarlo automáticamente.
- **kube-proxy:** es el proxy de red y balanceador de carga que implementa los servicios de red del cluster de kubernetes de cada nodo. Su función es enrutar el tráfico hacia el contenedor correcto.

Los objetos [29] que se despliegan en un cluster de kubernetes se pueden clasificar en dos grandes grupos. El primero son los objetos básicos:

El ***pod*** es la unidad desplegable más pequeña que se puede crear y gestionar en kubernetes. Puede contener uno o más contenedores *Docker*. El almacenamiento de datos puede ser interno y externo si se monta un volumen en la configuración. La red de los *pods* permite el intercambio y la comunicación de datos entre los contenedores que lo forman. Cada pod dispone de su propia IP y puerto. Concretamente, no es recomendable ejecutar más de un contenedor dentro de un *pod* para los cluster de kubernetes estables, debido a que es el objeto más efímero de kubernetes.

El ***Service*** [30]es el elemento que permite a los contenedores tener tráfico desde fuera del cluster. Los pods tienen una IP única, pero esta no es accesible desde fuera del cluster. Los dos tipos de servicios que se van a utilizar en este proyecto son los siguientes:

- *ClusterIP*: el servicio solo es accesible a nivel de cluster.
- *NodePort*: expone el servicio fuera del cluster mediante la IP del Master y un número de puerto configurable entre el 30000 y 32767.

Volume, [31] o volúmenes es una configuración que permite añadir un directorio de datos de un contenedor para que sea accesible desde fuera del *pod*. No se encuentra supervisado por kubernetes, es solo una configuración. De esta forma los datos quedan almacenados fuera en el cluster y si el *pod* se reinicia los datos se mantienen.

Los ***Persistent volumes*** (PV) es un elemento de kubernetes, se usan para administrar el almacenamiento duradero de un pod en el cluster. El ciclo de vida se encuentra administrado por kubernetes. Este elemento es necesario para despliegues productivos ya que permiten modificar el *pod* y eliminarlo, mientras que los *persistent volumes* siguen existiendo inalterables. Los *persistent volumes* se configuran mediante los ***PersistentVolumeClaim*** **(PVC)**. Estos son solicitudes de recursos en el cluster donde se detalla:

- El tamaño que se solicita al cluster para almacenar datos.
- El modo de acceso a la lectura y escritura de datos. Puede ser *ReadWriteOnce,* el volumen puede ser montado por un solo *pod* con permisos de lectura y escritura. Por otro lado, *ReadOnlyMany* que puede ser montado por un conunto de pods pero con permisos de lectura y *RaedWriteMany* donde el volumen puede ser accesible por varios pods con permisos de lectura y escritura.

Los *PersistentVolumeClaim* tienen el propósito de reservar los recursos necesarios en el cluster para un *persistent volume*. Si en el cluster existe un *persistent volume* que satisface esa configuración, se aprovisionan los recursos. Una vez se ha dado esta situación, el cluster vincula el *persistent volume* con el *pod* correspondiente.

Los ***Namespaces*** [32] sirven para crear separaciones lógicas dentro del cluster. Esta separación lógica se traduce en un cluster virtual de kubernetes. Se recomienda su uso si en el cluster participan un elevado número de usuarios y proyectos, ya que otra de sus funciones es la de dividir los recursos.

Los ***ConfigMaps*** permiten configurar variables de entorno, parametrizar despliegues o añadir configuraciones adicionales al contenedor que se despliega en el interior del *pod*.
El segundo grupo hace referencia a los objetos que tienen abstracciones de nivel superior, es decir, son los objetos básicos pero con una capa de funcionalidades adicionales.

Los ***Deployments*** [32] forman parte de los objetos de kubernetes que presentan funcionalidades adicionales. El deployment es un controlador, en el momento de la configuración se describe el estado deseado del *deployment*. Cuando se despliega en el cluster, el *deployment* se encarga de cambiar del estado actual al deseado de forma controlada. Se trata de una instancia de aplicación que se encuentra continuamente monitoreada. Si se da el caso de fallo, el controlador *deployment* crea nuevas instancias de *pod* que reemplazan las defectuosas. Para crear un *deployment* se necesita especificar la imagen *docker* y el número de réplicas que es necesario mantener en el cluster.

El ***Job*** forma parte de los objetos de kubernetes que presentan funcionalidades adicionales. El objeto tipo *job* se configura para realizar una tarea en concreto, por ejemplo, lanzar un comando o un *script*. Al inicio se despliega un *pod* que realiza la acción programada, una vez se ha completado la acción, el *job* pasa al estado completado y elimina los *pods* creados liberando los recursos.

Otros objetos que contienen funcionalidades adicionales son los DaemonSets, los ReplicaSet y los StatefulSet que no son utilizados en el código de este proyecto.

Para concluir, la plataforma de orquestación de microsiervos se encarga de gestionar el ciclo de vida de los contenedores de una aplicación. Los servicios que ofrecen van desde el manejo y auto-gestión del cluster, al *service discovery,* balanceo de carga, hasta al servicio de red. Ofrece servicios de monitorización de forma integrada como es el *dashboard* de Kubernetes.

4.2.2.4. Instalación de Kubernetes

La instalación de Kubernetes en las máquinas no ha sido de forma usual debido a las normas de seguridad de la empresa. Por lo que, la instalación que se detalla a continuación es óptima para servidores que se encuentran *offline.*

A parte de tener *docker* instalado, otro requisito fundamental es descargar los paquetes de herramientas de kubernetes en una máquina con conexión a Internet y subir los ficheros necesarios para la instalación a las máquinas virtuales.
Los paquetes de herramientas que se deben descargar son:

- kubeadm: la herramienta que permite crear el cluster de forma sencilla.
- kubelet: el componente de kubernetes que se ejecuta en los nodos workers y es el encargado de ejecutar los pods y los contenedores.
- kubectl: es la *command line interface* que nos va a permitir lanzar peticiones contra el cluster de kubernetes.

Sobre el directorio donde se tienen los ficheros descargados, se lanza este comando que ejecuta la instalación de kubernetes:

```
yum install -y --cacheonly --disablerepo=* *.rpm
```

Una vez finalizado este comando, se aplica el siguiente para ejecutar el kubeadm, que es la herramienta que permite desplegar el cluster de kubernetes de forma sencilla.

```
kubeadm config images list
```

Figura 11. Listado de imágenes del kubeadm.

A continuación, para seguir con la instalación se necesita descargar los ficheros de kubernetes relacionados con las conexiones del cluster. Se realiza el siguiente paso en una máquina con acceso a Internet.

wget https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml

Se descarga un fichero kube-flannel.yml, el cual contiene una línea donde viene especificada la imagen docker. Al tratarse de un cluster sin conexión a Internet, si se quiere hacer un despliegue, como en este caso de los *pods* kube-flannel, es necesario también descargar la imagen docker.

Se abre el fichero kube-flannel.yml y se inspecciona la línea *image,* se encuentra lo siguiente:

```
image: quay.io/coreos/flannel:v0.12.0-amd64
```

Con lo anterior, se conoce la imagen y la versión a descargar de *DockerHub*. Por último se guarda la imagen en formato comprimido *tar*.

```
docker pull quay.io/coreos/flannel:v0.12.0-amd64
```

```
docker save quay.io/coreos/flannel:v0.12.0-amd64 >
flannel_v0.12.0-amd64.tar
```

Se sube el fichero kube-flannel.yml y la respectiva imagen docker en formato *tar* a los servidores sin conexión. Por último se procede a descomprimir y generar la imagen mediante:

```
docker load  < flannel_v0.12.0-amd64.tar
```

Los pasos que se han realizado hasta este punto se consideran prerrequisitos de instalación del cluster. Para continuar con el proceso de instalación se necesita ser usuario privilegiado y deshabilitar la memoria swap:

```
swapoff -a
```

Se lanza lo siguiente para que SELinux se encuentre en modo *permissive:*

```
sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/'
/etc/selinux/config
```

Es necesario asegurarse de que el `config option sysctl` se encuentre con el valor de `net.bridge.bridge-nf-call-iptables` puesto a `1`. Para proceder con la comprobación:

```
cat <<EOF >  /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
sysctl --system
```

Opcionalmente se puede configurar el *command line interface* de kubectl para que sea autocompletable:

```
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

Finalmente, se comprueba si se ha instalado correctamente el CLI de kubectl:

```
kubectl version
```

Figura 12. kubectl version

Es preciso escoger una de las tres máquinas de las que se dispone como máster y realizar:

```
kubeadm init --pod-network-cidr=192.168.0.0/16 --kubernetes-
version=v1.18.2
```

Justo al finalizar este comando, se genera una salida que sigue la estructura de:

```
kubeadm join --token <token> <master-ip>:<master-port> --
discovery-token-ca-cert-hash sha256:<hash>
```

Es de vital importancia almacenar esta salida y guardarla de forma que pueda ser recuperada para ser utilizada más tarde en la configuración del cluster.

Continuando en la máquina virtual master, se verifica que el nodo master ha sido creado:

```
kubectl get nodes
```

La salida del comando anterior muestra que el estado del máster es NotReady ya que faltan las últimas configuraciones de red. Para ello, se continúa con:

```
grep -q "KUBECONFIG" ~/.bashrc || {
    echo 'export KUBECONFIG=/etc/kubernetes/admin.conf' >>
~/.bashrc
    . ~/.bashrc
}
```

De esta forma se permite la configuración del fichero `config kubernetes admin` que proporciona la configuración del comando kubectl como administrador de gestión del cluster.

En la misma ruta de archivos donde se encuentra el fichero kube-flannel.yml, se lanza el siguiente comando para desplegar los contenedores que permiten crear la red conexiones entre el nodo master y los *workers*.

```
kubectl apply -f  kube-flannel.yml
```

Pasados unos minutos, se tiene que realizar la comprobación siguiente:

```
kubectl get pods -n kube-system | grep coredns
```

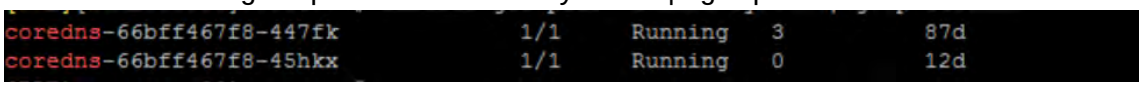


Figura 13. Pods coredns

El anterior comando es necesario realizarlo y comprobar que los *pods* de `coredns` se encuentran en estado *Running*. Coredns es el servicio DNS del cluster, resuelve y almacena en caché consultas de DNS.

Previo a establecer las conexiones de red entre el master y los nodos *workers*, es indispensable bloquear el posible despliegue y creación de *pods*, mediante:

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

Finalmente, es fundamental recuperar el comando anteriormente guardado y ejecutarlo, única y exclusivamente, en cada una de las máquinas workers.
El comando tiene una estructura similar a `kubeadm join --token <token> <master-ip>:<master-port> --discovery-token-ca-cert-hash sha256:<hash>`

Se recomienda esperar unos minutos, para luego ejecutar en la máquina master el siguiente comando:

```
kubectl get nodes
```

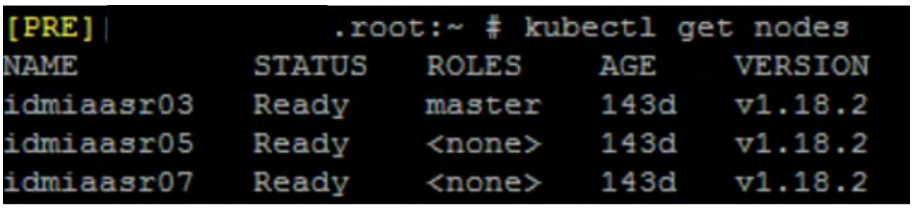


Figura 14. Nodos del master.

De esta forma se comprueba si se ha instalado de forma correcta el cluster de kubernetes. El estado de los nodos debe ser como el de la imagen X, *status Ready.*

4.2.2.4.1. Mitigación de re-arrancado de las máquinas virtuales

Para el correcto funcionamiento del entorno de pruebas y para mantener la estabilidad de los servicios desplegados es necesario disponer de una configuración que mitigue los re-arrancados de las máquinas virtuales.

Al reiniciarse cualquier máquina virtual esta es arrancada con la memoria *swap* por defecto. Los entornos de kubernetes no están preparados para funcionar con la memoria *swap* activada, más concretamente el kubelet no está diseñado para manejar ese tipo de memoria. Después de un reinicio de una máquina, el kubelet deja de funcionar y, en consecuencia, el cluster por completo.

El siguiente script permite desactivar la memoria *swap* automaticamente despues de un reinicio de las máquinas. El fichero llamado `init_script.sh`, se encuentra en el directorio `/home` de cada una de las máquinas y contiene las siguientes líneas de código:

```
#!  /bin/bash
swapoff -a
```

Para su ejecución en modo usuario *root*, se le proporcionan los siguientes permisos, primero se añade el usuario root como el propietario del fichero:

```
chown root.root /home/init_script.sh
```

A continuación, se da permisos de escritura sólo al usuario *root* y de ejecución para todos los demás usuarios.

```
chmod 4755 /home/init_script.sh
```

Por último, para que el *script* se ejecute cada vez que las máquinas se reinician, se debe modificar el fichero de `/etc/rc.d/rc.local` y se añade al final de todo, la ruta siguiente: `(/home/init:script.sh).`

Para concluir con el apartado, se ha preparado el entorno con tres máquinas virtuales con las dificultades que implican los servidores offline. El resultado ha sido un cluster de kubernetes que permite la orquestación de contenedores docker donde se van a desplegar las aplicaciones de la herramienta de este proyecto. En el siguiente apartado se detallan dichas aplicaciones y las tecnologías de las cuales se basan.

4.3. Desarrollo de la Herramienta

4.3.1. Diseño a alto nivel de la solución

A continuación, se explica la arquitectura técnica a alto nivel del *Back-end* así como el papel de cada componente.

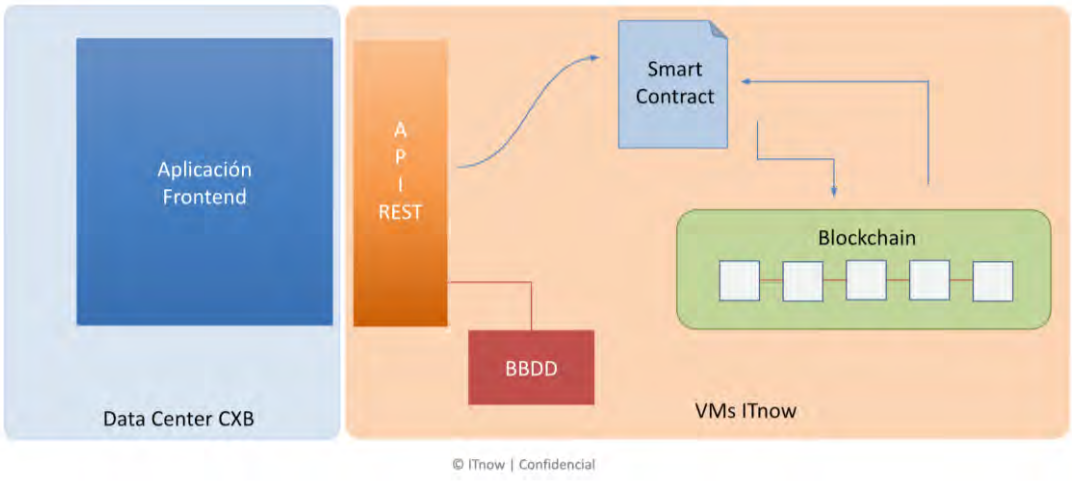


Figura 15. Arquitectura de la Aplicación con *Front-end* y *Back-end*.

Como se ve en la ilustración, el recuadro de color naranja hace referencia al Back-end que se implementa en esta primera fase. El ***Smart Contract*** es necesario para que la red ***blockchain*** siga la lógica de negocio. Este es imprescindible para saber qué guardar o leer de la blockchain, cómo hacer y si se puede hacer. A su vez, las funcionalidades de este *Smart Contract* son accesibles desde una API. La **API-Rest** es invocada por el *Front-end* para hacer acciones contra la *blockchain*. En la imagen, se observa que la API accede a una **base de datos** que se ha utilizado para guardar documentos de gran peso que por

definición y eficiencia no deberían ir en la *blockchain*, tales como las ofertas en PDF de las adendas.

4.3.2. Tecnologías utilizadas

Las tecnologías utilizadas para la implementación de la primera fase del proyecto del tipo Prueba de Concepto se presentan en el siguiente punto. Es necesario hacer la distinción de las tecnologías utilizadas para cada componente de la herramienta. La imagen de la figura 16 Arquitectura de la Aplicación muestra un resumen de las tecnologías implementadas.

- La red blockchain se implementa mediante el despliegue de *pods* de kubernetes que contienen contenedores *docker* proporcionados por *Hyperledger Fabric*.
- El smart contract, en *Hyperledger fabric* llamado *chaincode*, se ha escrito con el lenguaje de programación concurrente Go [34].
- La API-Rest se ha escrito con el lenguaje de programación NodeJS [35] y luego ha sido encapsulado en una imagen de *docker* para ser desplegado como un microservicio.
- La base de datos se ha implementado mediante la herramienta CouchDB [36]. Se trata de una base de datos no relacional que permite de forma ágil el almacenamiento de ficheros pesados.
- Plataforma del entorno de pruebas se ha implementado mediante *docker* y kubernetes. En primer lugar, todos los elementos que componen la aplicación están containerizados. Esto favorece y agiliza el desarrollo y el mantenimiento. En segundo lugar, kubernetes es un gestor de contenedores muy popular actualmente que ofrece muchas funcionalidades. Y, por último, existen entornos CaaS dentro de ITNow, de modo que la elección de kubernetes facilitará el traslado a *on-premises*.

Figura 16. Tecnologías utilizadas durante el proyecto faltan las versiones

4.3.3. Componente API - Rest

La API es un elemento que permite la interacción entre componentes y facilita la abstracción con capas inferiores de *software*. Por otro lado, REST es una interfaz que utiliza el protocolo HTTP para obtener o generar operaciones sobre datos.

La API REST desarrollada para este caso de uso permite la interacción a nivel de infraestructura de kubernetes con la red *Blockchain*, la base de datos CouchDB y la aplicación Frontal.

Hyperledger Fabric presenta un SDK programado en lenguaje NodeJS que permite realizar peticiones a la red. Para evitar los errores de incompatibilidad, la API REST ha sido programada en el lenguaje NodeJS.

Las operaciones que permite efectuar sobre la red son las relacionadas a cualquier sistema REST: POST, GET , PUT y DELETE.

4.3.3.1. Gestión del desarrollo API REST

En este punto se detalla el plan de trabajo y la gestión del desarrollo que se ha llevado a cabo para la implementación del componente API REST. A nivel de gestión se ha tratado como un subproyecto debido al gran número de dependencias entre el *chaincode* y la implementación del frontal por parte de la empresa externa.

Las tareas principales que se han identificado para coordinar los equipos de innovación y de desarrollo de la interfaz de usuario son:

- Análisis de los requisitos por parte del cliente y por parte de la red *blockchain*. Se especifican las funcionalidades finales del frontal que permitirán la edición de texto en la herramienta y adjuntar ficheros.
- Diseño cliente final del aspecto del frontal.
- Diseño funcional que se define junto con el flujo del proceso comercial y la arquitectura a nivel de código del frontal.
- Definición del modelo de datos que va a proporcionar el frontal a la herramienta basada en *Hyperledger fabric*. Es esta tarea se pacta el formato de la información y depende de la implementación del *chaincode*.
- Implementar el código relacionado con las comunicaciones con el frontal.
- Implementar el código relacionado con las comunicaciones y gestión de datos con la API y la base de datos.

La planificación del subproyecto API REST y Front-end se puede observar en el siguiente diagrama, la duración ha sido de 15 semanas.

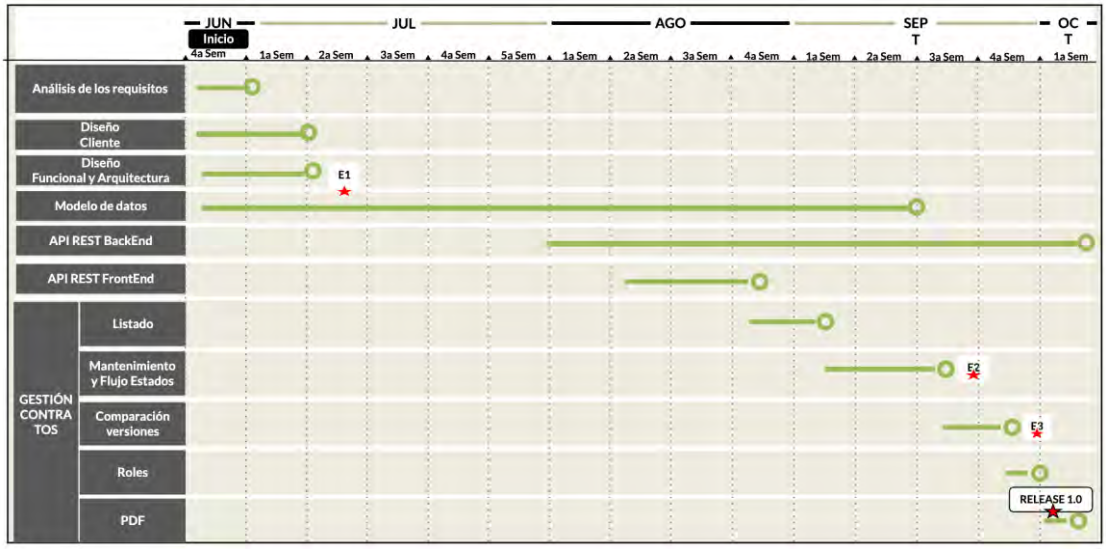


Figura 17. Planificación subproyecto API y Frontal

Los entregables por parte del equipo del *front-end* se detallan en la imagen mediante E1, E2, E3 y entrega final.

La entrega 1 (E1) es la documentación donde se detalla el diseño funcional, la arquitectura y el aspecto de la aplicación.

La entrega 2 (E2) es la entrega *alpha* del código frontal. A pesar de que se han ido realizando entregas de código para hacer las respectivas comprobaciones de conectividad, esta versión de la aplicación presenta los frontales del flujo de estados de la lógica de negocio.

La entrega 3 (E3) presenta la interfaz de usuario relacionada con el versionado del documento.

La entrega final consta de una aplicación web que permite la edición de los campos del documento comercial, mantener el seguimiento del estado de cada documento y el número de versiones. Adicionalmente permite la gestión y la visualización de la factura en formato pdf.

Las tareas que vienen detalladas en “Gestión de contratos” hace referencia a las tareas del equipo del frontal. Se necesitan tener en cuenta debido a que afectan directamente al desarrollo del código de la API y, en consecuencia, a la hoja de ruta establecida en el proyecto global por el equipo de innovación.

Las funcionalidades *front-end* y *back-end* vienen detalladas en la siguiente tabla. La intención de esta tabla es dar a conocer al lector la interacción entre la parte frontal y los elementos que componen el backend.

Tabla 5 . Funcionalidades *front-end* y *back-end*

Frontal web	*Back-end*
Se realiza la edición del documento comercial ya que ofrece la posibilidad de rellenar los campos necesarios de la adenda.	La API gestiona los datos del documento y los pasa a la red *blockchain*. Esta desencadena la ejecución del *chaincode* y se determina si los datos introducidos son correctos.
Presenta una pantalla para los múltiples versionados de un mismo documento.	Debido a la trazabilidad que aporta *blockchain* la api es capaz de proveer la información relativa a todas las versiones de un documento en cuestión.
Presenta pantallas diferenciativas en función del rol y del departamento que forma parte del proceso.	Mediante el conocimiento del estado del documento, la red conoce qué acciones pueden realizarse y por qué miembros. Las acciones de cada departamento están escritas en la logica del *chaincode*.
Permite subir el documento tipo factura en formato PDF relativo al proceso comercial.	La API gestiona el documento tipo factura, haciendo un cambio de formato a base64 y realizando un *hash*. De esta forma, se almacena el PDF en base64 en la base de datos. Por otro lado, se produce en la *blockchain* una transacción para almacenar el *hash* del documento en la ledger. En consecuencia, se vincula la factura con el proceso comercial. Para recuperar el documento, primero se hace una transacción a la *blockchain* y se pide el *hash* del documento. La API compara el *hash* que tiene almacenada la *ledger* contra el *hash* del documento que está almacenado en la base de datos. Esta implementación asegura que el documento guardado fuera de la red *blockchain*, es decir, el documento PDF almacenado en la base de datos, no ha sido modificado en ningún momento.

4.3.3.2. Implementación API - REST

El desarrollo de este componente en particular ha sido parcialmente implementado por la persona que redacta el documento.
La conexión con el frontal se realiza mediante el protocolo HTTPS. La autenticación es *Basic Access Authentication* que provee credenciales en forma de usuario y contraseña. De esta forma se securiza la conexión entre el *Front-end y el Back-end*.

Los procesos de implementación de la API-REST se pueden dividir en desarrollo del código NodeJS, registro de la API como elemento de la blockchain, contenerización del código y despliegue del *deployment* en kubernetes.

El código NodeJS que construye la API-REST se encuentra estructurado en los siguientes módulos:

Tabla 6. Estructura código API-REST

package.json	Enumeración de todos los *packages*, librerías y dependencias para la ejecución del servidor.
package-lock.json	Generado automáticamente al ejecutar el package.json, contiene información sobre las dependencias y sus versiones.
server.js	Inicialización del servidor en el puerto indicado, definición de variables y constantes, gestión de la Autenticación y cifrado del protocolo HTTPS
routes.js	Definición de las rutas y los métodos. Nexo de conexión entre la red *blockchain* y la base de datos CouchDB.
redFabric.js	Inicialización de la red *blockchain*, comprobación del usuario Admin, métodos que permiten hacer las queries *smartcontract*.
FabricClient.js	Métodos para realizar transacciones y *queries* en la *blockchain*.
registerAdmin.js	Genera las *Certificate Authorities* de carácter Admin que permiten a la API realizar queries como si fuera un elemento más de la red *blockchain.*
registerUser.js	Genera las *Certificate Authorities* de carácter User que permiten a la API realizar queries como si fuera un elemento más de la red *blockchain.*
utils.js	Métodos que comprueban los datos introducidos desde el frontal y decodifican de base64 a texto.

La API de una red *blockchain* necesita poderes de administrador en la red, para poder ejecutar transacciones y realizar consultas al *Chaincode* como si se tratara de un participante más.

Primero de todo hay que configurar el fichero “connection-profile.yaml” con toda la información de la red. Este fichero se encuentra disponible en *Hyperledger Fabric SDK* y ofrece los diferentes *connection profiles* que describen el conjunto de componentes que forman la red (*peers*, *orderers* y *Certificate Authorities*). Este *connection-profile* es utilizado por la API para configurar una puerta de enlace (*gateway*) que tiene la capacidad de interactuar con el canal sin preocuparse de la topología de red. Se entiende por topología al conjunto de elementos que componen la red.

En conclusión, el fichero proporciona a la API el conocimiento de la configuración de la red *blockchain* y le otorga poderes para realizar transacciones.

La imagen de docker de la API se crea a partir de un Dockerfile que contiene la imagen base `node:8.16-alpine`. Esta imagen contiene el sistema operativo alpine y NodeJS instalado. Para concluir con el Dockerfile de la imagen customizada, se importan los ficheros con el código y se lanza el comando `npm run start` para levantar el servidor NodeJs. El Dockerfile en cuestión se encuentra en los documentos anexos, en el Anexo 2.

La implementación del objeto *deployment* de kubernetes para la API-REST presenta las características que se pueden ver en la tabla:

Tabla 7. Parametros Deployment de la API-REST

Service	
Nombre del servicio	apitest
Namespace	api-namespace
NodePort	30080
Port	8080

Pod	
Tipo	Deployment
Nombre	apitest
Namespace	api-namespace
Imagen	Creada y customizada con el Dockerfile
Command	"/bin/bash", "-ec", "while :; do echo '.'; sleep 50 ; done"
ContainerPort	8080
Volumes	nfs/apiRestfulDir

El hecho de que se trate de una prueba de concepto y el entorno *cloud* sea de pruebas no es necesario implementar *persistent volumes* y *persistent volumes claim.*

4.3.4. Componente Base de Datos

La base de datos utilizada en la fase primera del proyecto en estado de prueba de conceptos es CouchDB, se trata de una base de datos NoSQL que permite guardar con facilidad JSON.

La CouchDB ha sido pensada para almacenar información que no necesita ser gestionada con las características de la *blockchain*, tanto por su tamaño o por su impacto en la lógica

del proyecto. Se han implementado dos listas ordenadas, también llamadas tablas para almacenar dicha información: tabla de adendas y tabla de ofertas.

- Adendas: se guardan los atributos del proceso comercial que no es necesario que exista un control de la trazabilidad e inmutabilidad, por lo que se guardan fuera de la red *Blockchain*.

Tabla 8. Datos tabla adendas CouchDB

Datos tabla adendas CouchDB
Receiver: Receptor de la adenda
Objective: Comentario del Objetivo de la adenda
Period: Perido (**startDate** & **endDate** or numberOfMonths)
Charges: Cargos (**concept:** concepto se compra & **amountConcept:** dinero por concept)
BillingAndPayment: Facturación y pago - (**Description:** es parecido al concepto, **billingDate:** Fecha de facturación, **paymentDate:** Fecha de pago, **bill:** importe sin impuestos)
AdditionalService: Servicio adicional que será siempre 0
PDFSOFileName: Nombre del archivo PDF de la oferta del proveedor
Párrafos modificados: JSON array de los Párrafos modificados y su contenido.

- Ofertas: se guarda las ofertas de la adenda en formato Base64.

Tabla 9. Datos tabla ofertas CouchDB

Datos tabla ofertas de la CouchDB
PDFSOFileName: Nombre del archivo PDF de la oferta del proveedor
pdfBase64: pdf codificado a base 64

4.3.4.1. Implementación Base de Datos

La base de datos CouchDB está implementada con el objeto de kubernetes del tipo deployment que genera un *pod*. También se implementa un objeto tipo servicio de kubernetes. Los datos guardados se encuentran en el Network File System del cluster ya que no es necesario un *persistent volumes*.

La implementación para la CouchDB presenta las características que se pueden ver en la tabla 10:

Tabla 10. Parametros deployment couchDB

Server CouchDB entorno Desarrollo	
Nombre	couchdb-service
Protocolo	TCP
Port	5984
NodePort	30006
Pod de CouchDB	
Nombre	couchdb
Namespace	couchdb
Imagen de Docker Hub	couchdb:2.3.0
Persistencia de Datos	NFS

4.3.5. Componente Hyperledger Fabric

En este punto se describe el proceso de implementación y diseño de la arquitectura de la red *blockchain Hyperledger Fabric* sobre el orquestador de contenedores kubernetes.
Con la intención de definir la etapa de diseño del *chaincode* se hace énfasis en la lógica de negocio del proceso comercial. De esta forma, el lector va a ser capaz de comprender mejor los estados por los que pasa el proceso de adendas y la razon de la topologia de la red *blockchain*.

El componente de la herramienta basado en *Hyperledger Fabric* consta de la infraestructura de la red *blockchain* y el *chaincode*. Primero, se explica el diseño y el desarrollo del *chaincode*, para seguidamente explicar la arquitectura cloud de la red *hyperledger*.

4.3.5.1. Diseño chaincode

El *chaincode* proporciona a la red *blockchain* la capa de negocio. La lógica que se quiere plasmar es la explicada en el punto 4.2.1
Con el objetivo de materializar el flujograma del proceso de adendas comerciales en código ha sido necesario crear el siguiente diagrama de estados:

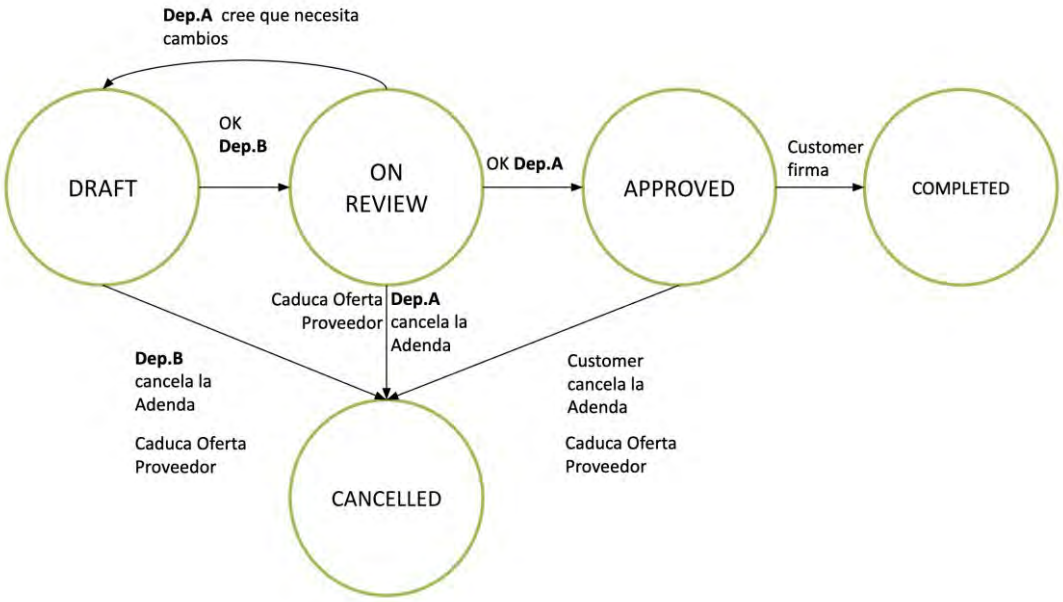


Figura 18. diagrama de estados del proceso.

Los estados por los que pasa el documento se describen de la siguiente forma:

- En el estado **Draft** se crea un nuevo registro de adenda en la red *blockchain* o la adenda ha sido devuelta por departamento A para que se acabe de revisar para poder validarla. Se necesitará dar el OK (por departamento B) para que pase a revisión por el departamento A.
- En el estado **On review** la adenda necesita la aprobación y/o revisión por el departamento A. No se puede modificar la adenda.
- En el estado **Approved,** la adenda ha sido aprobada por el departamento A y está a la espera que el customer la firme.
- El estado **Completed** es en el que la adenda ha sido firmada y el proceso ha terminado; la adenda está activa hasta la fecha indicada. Cuando queden X días para su expiración, se lanzará una alerta desde la *blockchain* notificando que la adenda en cuestión está a punto de expirar.
- **Cancelled**, La adenda ha sido cancelada y finaliza el proceso.
    - Expiración de la oferta del proveedor.
    - Cancelada por el departamento B.
    - Cancelada por el departamento A.

De lo anterior explicado junto con las acciones que puede realizar cada departamento sobre el documento, se extraen los siguientes roles que también van a ser plasmados en el *chaincode*:

- El rol **creador** permite crear, modificar y cancelar el proceso. Corresponde al departamento B.
- El rol **revisor** permite aprobar y cancelar el proceso. Corresponde al departamento A.
- El rol **lector** permite ver y consultar el estado del proceso. Corresponde al grupo *customer*.

Adicionalmente de cumplir con la lógica de negocio, el *chaincode* tiene que programarse con los requerimientos adicionales que se han pedido por parte del cliente final que utilizará la herramienta. Es decir, el *chaincode* tiene que plasmar los siguientes requisitos:

- Almacenar los datos sensibles en la *blockchain*.
- Gestionar las ID referentes a la información almacenada fuera de la blockchain en el couchDB. Esto incluye el PDF con el documento tipo factura y los datos correspondientes a una adenda que no requieren trazabilidad.
- Cancelar la addenda si el fichero adjunto tipo factura se encuentra caducado.

El *chaincode [37]* implementa los requisitos y las funcionalidades mediante médotos, estos siguen una estructura marcada y vienen proporcionados por la **interficie *Chaincode***. La interfaz incluye varias funciones por defecto para interactuar con las transacciones y gestionar el chaincode:

El método **init** inicializa el *chaincode*. Este método se encuentra en todos los *chaincodes* ya que es un requisito indispensable. Dada su finalidad, este método nunca se ejecuta por defecto, si no que tiene que ser llamado en el inicio de la red *blockchain* para iniciar la ejecución del *chaincode*.

El método **invoke** es particular de la interfaz *chaincode* y se llama en respuesta a la recepción de una transacción de *invoke* para procesar las propuestas de transacción. En otras palabras, este método es ejecutado si se lanza una propuesta de transacción a la red *blockchain*. Este método puede utilizarse para comprobar el estado de la *ledger* como para actualizarlo.

Por otro lado, tenemos la interfaz **ChaincodeStubinterface**, más conocida como ***shim***. La interfaz *shim* provee de varias APIs para gestionar el acceso y la modificación del estado de la ledger.

Las funciones principales de shim son **GetState**, que sirve para obtener información de la ledger sobre un componente en concreto, y **PutState**, que sirve para escribir un nuevo componente en la *ledger*, en otras palabras, modificar el estado.

Las funciones utilizadas que se consideran imprescindibles para del código de un *chaincode* son:

- stub.GetState(componente): permite obtener la información en la *ledger* sobre un componente
- stub.PutState(componente): permite modificar el componente almacenado, es decir, sirve para escribir el componente en la *ledger*.
- shim.Error: detiene la ejecución del *chaincode* y notifica de un error. Normalmente se emplea para programar el flujo de estados.
- shim.Success: finaliza la ejecución del *chaincode* y comunica el resultado de la ejecución.

En el Anexo 3 se encuentra el código del *chaincode* con los métodos necesarios para que la red opere en función de los requisitos de funcionalidad que se pide de la herramienta de adendas. Se encuentra un breve resumen de los métodos y del código completo.

4.3.5.2. Diseño Red Hyperledger Fabric Fase 1

El diseño de la red *hyperledger* para este caso de uso ha sido planteado desde la perspectiva de prueba de concepto. En esta fase del proyecto se quiere demostrar que es posible desplegar la solución *hyperledger fabric* versión 1.4 en kubernetes. En consecuencia, los componentes y la arquitectura blockchain se plantean en modo test.

La red *hyperledger* está formada por organizaciones, en este caso dos, el departamento A y B. Cada organización se le proporciona un *peer* de tipo *endorser*. Los endorser peers son capaces de recibir una invocación de petición de transacción y, en función de la ejecución del *chaincode*, aprobar la transacción y actualizar la *ledger*. Los peers necesitan mantener la *ledger* medinate couchDB, por lo que en esta arquitectura habrán dos componentes de este tipo.
Se proporciona un *chaincode* a cada organización para que puedan determinar las transacciones.
A continuación siguiendo con el diseño, se define el servicio de ordenación que garantiza la consistencia de las *ledgers*. Desde una perspectiva de pruebas, se opta por la implementación del nodo *orderer* en Solo. Esta configuración presenta una única instancia de *orderer*.
Las organizaciones en la red *blockchain* van a tener permisos de *Admin* en la configuración MSP, esto significa que tendrán acceso de administrador al canal para realizar transacciones. Las credenciales que van a representar las identidades de cada organización van a ser mantenidas por el componente CA. Ambas organizaciones tendrán su propia CA.
Se establece un único canal de comunicación entre las dos organizaciones donde se lanzarán las transacciones.

Por último, se ha pensado el componente CLI que permitirá montar y configurar la red *blockchain* y gestionar el despliegue del *chaincode*.

La infraestructura diseñada en este punto no está pensada para entornos productivos y, en consecuencia, no es tolerante a fallos.
El diseño explicado en este punto se encuentra de forma visual descrito en la siguiente figura 19.

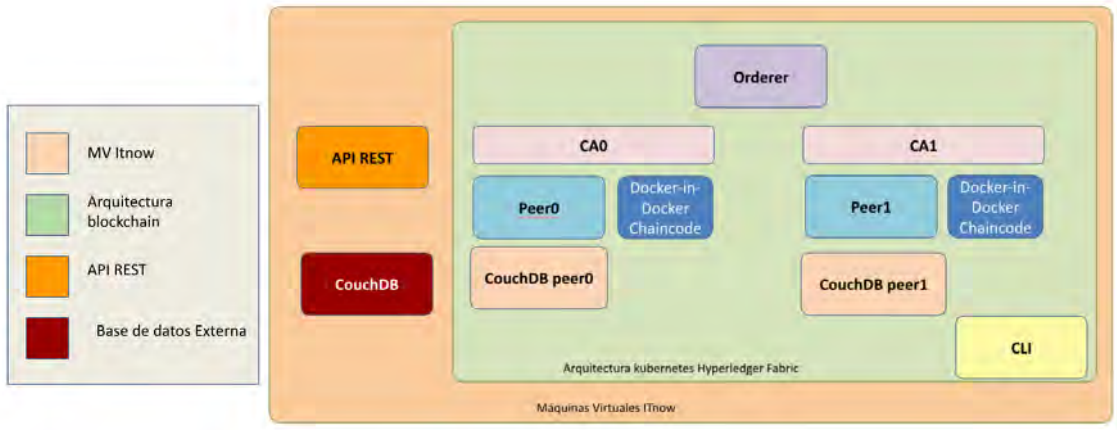


Figura 19. Diagrama *Back-end* con la red *Hyperledger Fabric* v1.4 implementada en kubernetes

4.3.5.3. Implementación de la red *Hyperledger Fabric*

La implementación de cada uno de los componentes en kubernetes se hace mediante objetos del tipo *deployment* que tienen como base las siguientes imágenes docker.

Tabla 11. Imagenes Hyperledger Fabric version 1.4.1

POD	Docker Imágenes
Peer	hyperledger/fabric-peer:amd64-1.4.1
Orderer	hyperledger/fabric-orderer:amd64-1.4.1
Certificate Authority	hyperledger/fabric-ca:amd64-1.4.1
CouchDB-peer	hyperledger/fabric-couchdb:amd64-0.4.14
CLI	hyperledger/fabric-tools:amd64-1.4.1
Docker-in-Docker Chaincode	docker:stable-dind

Se puede ver en la tabla que el proyecto *Hyperledger Fabric* distribuye su solución mediante contenedores Docker. Por lo que su solución está optimizada para ese entorno. Una muestra de ello es la instalación del *chaincode*.

El código base de la imagen del *peer* está programado de tal forma que, cuando se instala el *chaincode* por primera vez en la red hyperledger, se genera un contenedor *docker* con el fichero del *chaincode* (chaincode.go) en ejecución. En otras palabras, el *peer* es el

encargado de generar la imagen del *chaincode* con el fichero chaincode.go y crear un contenedor docker con este fichero ejecutándose.

Para instalar el *chaincode* en kubernetes el deployment del *peer* debe tener la siguiente configuración:

```
volumeMounts:
- mountPath: /host/var/run/docker.sock
  name: peer0A-adendas-claim0
- mountPath: /etc/hyperledger/fabric/msp
```

Como se ve en el código, se configura el *volume* del *pod* del *peer* con el docker.socket. Es decir, la configuración `/host/var/run/docker.sock` permite al contenedor del *peer* lanzar peticiones directamente sobre la API de docker.

Dicho de otra forma, el docker.sock es el *socket* de dominio UNIX donde el Docker daemon realiza las escuchas, por lo que el *peer* tiene acceso directo a este proceso. Esta implementación no es deseable para ningún entorno de docker. Supone un gran agujero de seguridad ya que existen dos *pods* en el cluster, dos *peers*, que tienen acceso al docker.sock, es decir al Daemon del cluster, y pueden crear contenedores docker sin la supervisión de kubernetes.

Para solucionar este agujero de seguridad y tener controlado el *chaincode* bajo la orquestación de kubernetes, se ha ideado la implementación Docker-in-Docker.

El Docker-in-Docker es un docker daemon corriendo dentro de un contenedor de docker. Este componente se puede desplegar en kubernetes.

Ahora el *peer*, en vez de escuchar el *docker deamon* del cluster, va a escuchar el *docker daemon* que se encuentra dentro de un contenedor docker supervisado por kubernetes.

El flujo de despliegue del *chaincode* se comenta en la figura 20.

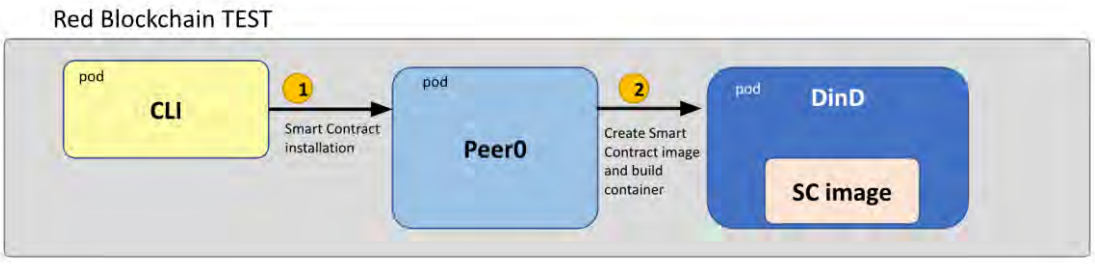


Figura 20. Flujo de despliegue *chaincode* con DinD. SC Image es la imagen docker del *chaincode*.

El comando para instalar el *chaincode* se lanza desde el CLI para el *peer* (1). El *peer* que tiene montado el docker.sock del Docker-in-Docker en la configuración de los volúmenes, es capaz de lanzar peticiones contra el *Docker daemon* del interior del pod DinD. El peer ordena al docker daemon generar la imagen del *chaincode* y levantar un contenedor docker con esa imagen.

De esta forma tenemos un contenedor docker con el código del *chaincode* corriendo supervisado por kubernetes.

La solución alternativa ofrece al *chaincode* tener las características de monitorización y gestión del ciclo de vida que proporciona kubernetes.

En este punto se explica solamente el ciclo de instalación e instanciación del *chaincode* para la versión 1.4 de *Hyperledger Fabric* ya que es el que aporta un valor significativo en el proyecto en fase *Proof of Concept*.

El despligue por completo de la red *Hyperledger Fabric* se realiza en el siguiente apartado. Se ha decidido de esta forma ya que, en dicho apartado se despliega la última versión de *Hyperledger Fabric*, con el diseño de los componentes en versión productiva.

4.3.5.4. Resultados de la Fase 1 de la Prueba de Concepto

Los resultados de esta implementación han sido **satisfactorios**. La simplicidad del diseño con dos organizaciones, un solo canal y un *chaincode* instanciado en cada peer ha supuesto el perfecto escenario de pruebas para innovar una opción de despliegue del *chaincode* en un entorno de kubernetes.

Es decir, el propósito de la Prueba de Concepto está cumplido. Se nos pidió: diseñar, gestionar e implementar una PoC basada en *Hyperledger Fabric* que permita la optimización de un proceso comercial.

La herramienta integrada por el frontal web y el *backend* basado en *blockchain* ha sido testado durante meses por parte del cliente y del departamento de mejora contínua de la empresa. Los dos departamentos implicados en el proceso aseguran que el proceso comercial de adendas ha sido mejorado:

Tabla 12. Resultados Prueba de concepto y ventajas

Sin la herramienta de adendas	Con la herramienta de adendas	Ventajas
Las comunicaciones y el flujo del proceso tiene que hacerse mediante correo electrónico a un buzón genérico de los departamentos .	El flujo del proceso viene marcado de forma automática mediante la red *blockchain* y las comunicaciones se realizan a través de comentarios en la página que se escriben en la *ledger*.	Se conoce el estado de la adenda en cuestión y qué departamento tiene
El documento tipo factura se adjunta al principio del proceso vía email y es	La factura se guarda off-chain y queda vinculada al proceso comercial en la	La herramienta proporciona un listado de todas los procesos

responsabilidad de cada departamento mantener la coherencia entre proceso comercial y factura adjunta.	*blockchain*.	comerciales que están en curso y el tiempo que falta para que se caduque el proceso.
El gran número de revisiones y modificaciones por parte de los dos departamentos dificulta conocer la última versión del documento.	Registro de todas las versiones y modificaciones realizadas al documento.	Absoluta trazabilidad de los estados del proceso. Valor añadido de la seguridad. Solo se puede modificar el documento las personas con los roles autorizados.

Debido al resultado satisfactorio de la prueba y de la mejora del proceso, los departamentos implicados han querido llevar a producción la herramienta.

A continuación se explica el proceso de adaptación de la herramienta para estar en producción.

4.4. Puesta en producción de la herramienta

El objetivo de la **segunda fase del proyecto** es adaptar los elementos y el código entregado en la Prueba de Concepto para poder ser desplegado en el entorno *cloud* bancario privado. En este punto se describirán las acciones necesarias para la migración del *back-end* del entorno test de laboratorio a la infraestructura corporativa.

4.4.1. Validación Diseño

En el presente punto se analizan los componentes de la herramienta desarrollados por parte del departamento de Innovación uno a uno y se validan con el departamento de infraestructura de ITnow. A su vez se analizan las vulnerabilidades y la viabilidad de migración al entorno corporativo de cada componente.

4.4.1.1. Componente API-REST

El componente API-REST está formado por el código escrito en NodeJS y la implementación en kubernetes.

La parte relativa al código de NodeJS se le realizan pruebas de estrés y de control de errores. Estas son pasadas satisfactoriamente.

La implementación en kubernetes tiene que cumplir lo siguiente:

- La imagen docker de la API no puede basarse en la imagen *open source* `node:8.16-alpine`. La imagen tiene que ser la del repositorio privado de caixabank `rhel8/nodejs-10`, con la condición de cargar en la imagen todos los módulos de *node* necesarios para la ejecución del código *node*.
- El contenedor creado con la imagen no puede ejecutarse en *root*. Al realizar el cambio a la base imagen *node Redhat*, esto no supone un problema ya que esta imagen por defecto se ejecuta con un usuario default.
- Disponer de memoria persistente garantizada por los objetos de kubernetes, *persistent volumes*.

El Dockerfile de la imagen de la API-REST se puede encontrar en el Anexo 4. El *deployment* junto con los *persistent volumes* y los *persistent volume claim* se encuentran en el Anexo 5.

4.4.1.2. Componente CouchDB

El componente CouchDB, la base de datos externa del *back-end*, no puede ser desplegado en la infraestructura de la empresa por tres razones.

- Las bases de datos no se despliegan en formato kubernetes en el *cloud* on premise.
- Las bases de datos de la empresa no pueden ser no relacionales.
- Las bases de datos tienen que estar implementadas en la infraestructura SQL empresarial.

La política de la empresa de estandarizar el formato de los datos, simplificar el sistema de *back-up* y de recuperación de los datos en caso de fallo, obliga a cambiar el componente *cloud* couchDB por una solución SQL empresarial.
Este cambio implica modificar los métodos y funciones de la API-REST encargados de gestionar los datos *off-chain*. Ahora estos métodos tienen que operar y almacenar los datos siguiendo la forma de operar de esta base de datos que es mediante tablas.

4.4.1.3. Componente Hyperledger Fabric

El equipo de infraestructura de ITnow y seguridad *Caixabank* revisaron con bastanta profundidad los elementos de Hyperledger Fabric en busca de vulnerabilidades a nivel de seguridad, incumplimiento de requisitos y del código de buenas prácticas de *cloud* elaborado por la empresa.

Los dos departamentos concluyeron lo siguiente:

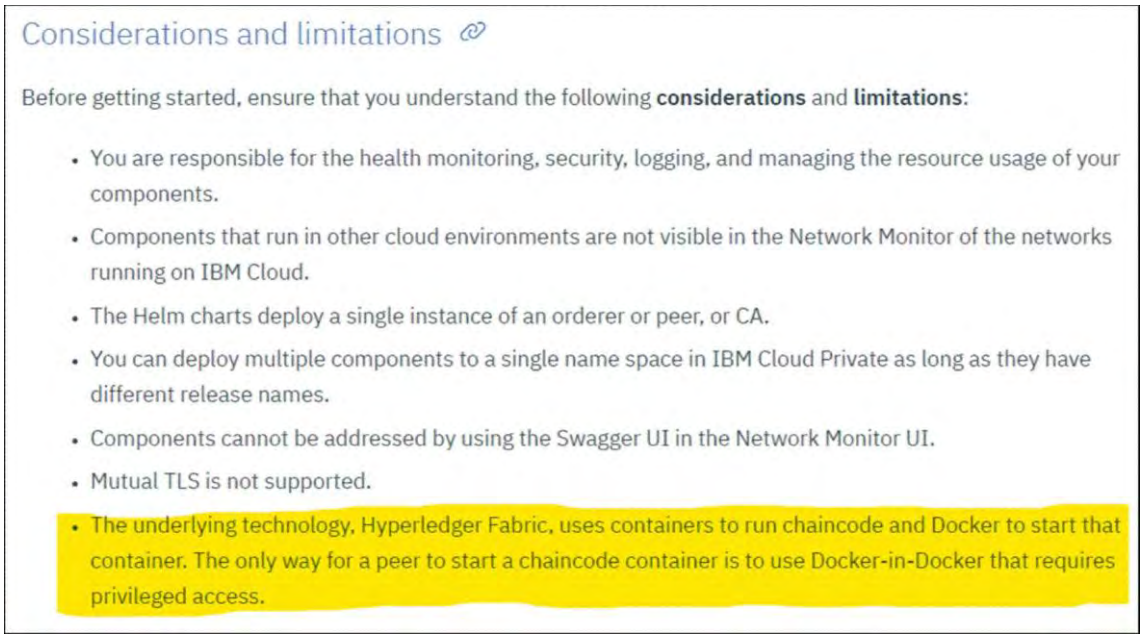
- Los contenedores de Hyperledger Fabric tienen que ejecutarse todos en usuario *rootless*. Las imágenes es recomendable que sean base *RedHat*.
- Se permiten los pods de CouchDB al tratarse de un requisito impuesto por el fabricante del producto, Hyperldger Fabric.
- No se permite que el *pod* del peer tenga montado el *docker.sock* en la configuración de los *volumes*.

- La solución Docker-in-Docker no puede ser implementada de ninguna forma en el cluster productivo *on premise*. Debido a la brecha de seguridad que implica tener un *pod* que tiene total acceso a un *docker daemon*. Por otro lado, la normativa de la empresa no permite que se ofrezca un servicio en máquinas de test o de laboratorio.

De acuerdo a los puntos anteriormente mencionados, la red Hyperledger Fabric no es viable de ser implementada en la empresa. Para tratar de solventar estos inconvenientes, desde el equipo de innovación surgen dos enfoques:

1. Intentar la aprobación por parte de seguridad CaixaBank aportando una justificación o confirmación por parte de la comunidad Hyperledger o de IBM Hyperledger Fabric, explicando que la única solución para desplegar una red Hyperledger Fabric con la última versión estable es mediante *DinD*.
2. Tratar de modificar una imagen de *DinD*, por ejemplo, `docker:19.03-dind`, para que el docker daemon de su interior no sea ejecutado en *root*. Además, este solo puede atender las peticiones lanzadas por el *pod* del peer y de ningún otro elemento del clúster.
3. Implementar una red en el entorno test idéntica a la productiva para crear las imágenes del *chaincode* en el entorno de pruebas, guardar esas imágenes y subirlas al repositorio de imágenes privado de CaixaBank.

El punto 1. IBM Hyperldger Fabric confirma la limitación existente por parte del código de Hyperldger Fabric y que la única implementación que permite el despliegue del *chaincode* en kubernetes es mediante el *workaround* de Docker-in-Docker. La justificación se encuentra en la siguiente figura:



Considerations and limitations

Before getting started, ensure that you understand the following **considerations** and **limitations**:

- You are responsible for the health monitoring, security, logging, and managing the resource usage of your components.
- Components that run in other cloud environments are not visible in the Network Monitor of the networks running on IBM Cloud.
- The Helm charts deploy a single instance of an orderer or peer, or CA.
- You can deploy multiple components to a single name space in IBM Cloud Private as long as they have different release names.
- Components cannot be addressed by using the Swagger UI in the Network Monitor UI.
- Mutual TLS is not supported.
- The underlying technology, Hyperledger Fabric, uses containers to run chaincode and Docker to start that container. The only way for a peer to start a chaincode container is to use Docker-in-Docker that requires privileged access.

Figura 21. Captura de una documentación del producto IBM Blockchain Platform, describe las consideraciones y limitaciones de Hyperldger Fabric por parte de IBM.

Los departamentos de Caixabank Seguridad Informática, Seguridad ITnow e Infraestructura *Cloud* ITnow no aprobaron el Docker-in-Docker como requisito de producto.

El punto 2. Se realiza la prueba con la imagen de `docker:19.03-dind`. Docker presenta una serie de Dockerfiles y de manuales para conseguir configurar un usuario que ejecute los contenedores y el docker daemon como usuario *rootless*. Estos manuales e implementaciones proporcionadas por Docker están aún a día de hoy en estado experimental.

Desde el departamento de innovación se realizaron varias pruebas y ninguna con éxito debido a que el despliegue del pod DinD con estas modificaciones no era estable. No se mantenía levantado y dejaba de funcionar a los pocos minutos.

Se puede comprobar el Dockerfile del DinD en el Anexo 6.

El punto 3. Es el que se había considerado como una posible solución alternativa, pero la imagen del *chaincode* generada dentro del DinD, no puede ser implementada en estado rootless una vez creada.

4.4.2. Estudio de la red Hyperldger Fabric v2.0.1

La falta de madurez del código de Hyperledger Fabric v.1.4

La versión de Hyperledger Fabric v2.0.1, lanzada el 26 de febrero de 2020, presenta las siguientes características:

- El *external chaincode launcher.* Elimina la dependencia con el *docker daemon* y permite desplegar el *chaincode* de la red *blockchain* en cualquier entorno.
- Los *External Builder Exceutables*. Son un conjunto de *scripts* que cargan la configuración del *chaincode* en el peer y ejecutan el chaincode.
- El chaincode como *External Service*. En esta versión el *chaincode* está pensado para estar desplegado dentro de un *pod* de kubernetes y funcionar como un servicio totalmente externo e independiente al *peer*.

Una vez salida la *release* de Fabric 2.0.1, se retoma la **fase dos del proyecto** de la herramienta corporativa de las adendas.

Durante tres semanas se realiza un estudio exhaustivo de las nuevas características de la nueva versión, y se despliega la primera *blockchain* en kubernetes. Se comprueba que la red sigue funcionando como antes, pero al probar la nueva forma de desplegar el *chaincode* no funciona debidamente.

Siguiendo con el desarrollo, se toma consciencia de que varios desarrolladores alrededor del mundo tampoco pueden desplegar el *chaincode* mediante lanueva característica externeal *builders*.

Para seguir avanzando en el desarrollo de la solución y realizar la instalación del *chaincode* mediante kubernetes se recurre a Hyperledger Fabric Project. Uno de los desarrolladores

se sincera de que la función external builders no se encuentra en fase estable y se presta para ayudar en el desarrollo de esa función.

Tras estar bastante tiempo realizando pruebas se consigue desplegar un *chaincode* en el orquestrados de kubernetes sin la necesidad de docker Daemon.

4.4.3. Diseño a alto nivel de la solución Productiva

En este apartado se presenta el diseño a alto nivel de la herramienta de adendas con los cambios anteriormente comentados.

El diseño de la herramienta después de la validación de diseño por parte de Seguridad y Infraestructura de la empresa presenta el siguiente esquema:

- La base de datos couchDB se cambia por el servicio SQL *on premise*.
- Se modifican los métodos de la API-REST que hacen referenica a los datos off-chain.
- Se implementa el diseño de la red Hyperledger Fabric para ser desplegada en el entorno *cloud* de la empresa.

Se procede al despliegue de esta solución en los entornos *cloud* privado test, pre y pro.

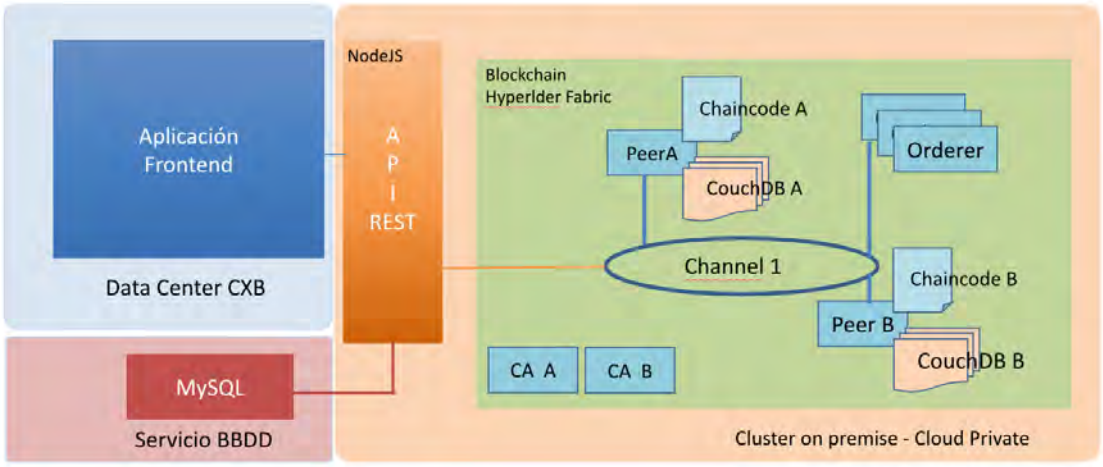


Figura 22. Herramienta del proceso comercial *on premise*.

4.4.4. Diseño de la red Hyperldger Fabric v2.0.1

En este apartado se presenta el diseño de la red Hyperledger fabric con la versión 2.0.1 para el caso de uso de la herramienta de adendas.

El diseño de la red y el número de componentes varía respecto la red de la Prueba de Concepto ya que se pretende desplegar.

Los requisitos marcados por el código de buenas prácticas, la seguridad y los planes de contingencia en caso de desastre junto con el caso de uso han originado el siguiente diseño de red *blockchain*.

La topología de la red *blockchain* escogida es la siguiente:

- Dos Organizaciones: Una para cada departamento.
- Un canal de comunicación.
- Dos *peer endorser,* uno para cada organización.
- Dos *couchDB* para mantener la ledger de cada *peer.*
- Dos *chaincodes*, uno para cada *peer.*
- Dos API-REST.
- Tres *Orderers* con el tipo RAFT como servicio de ordenación.
- Dos Certificate Authority.

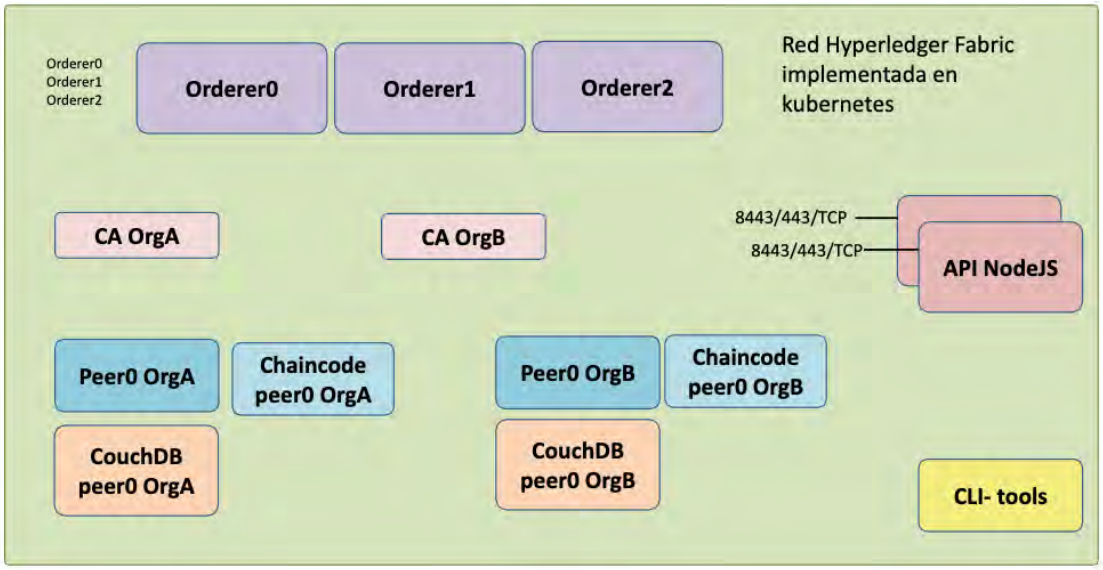


Figura 23. Muestra el número de componentes de la red Hyperldger Fabric escogidos.

4.4.5. Implementación de la red Hyperledger Fabric on-premise

La implementación de cada uno de los componentes en kubernetes se hace mediante objetos del tipo *deployment* que tienen como base la siguientes imágenes docker:

Tabla 13. Imágenes de Hyperledger Fabric 2.0.1

POD	Imágenes Docker
Peer	hyperledger/fabric-peer:amd64-2.0.1

Orderer	hyperledger/fabric-orderer:amd64-2.0.1
Certificate Authority	hyperledger/fabric-ca:amd64-2.0.1
CouchDB-peer	hyperledger/fabric-couchdb:amd64-0.4.14
CLI	hyperledger/fabric-tools:amd64-2.0.1
Chaincode	rhel8/go-toolset:1.12.8

El código de los componentes de kubernetes debe seguir las pautas marcadas por el código de buenas prácticas:

- **Imágenes rootless**

El desarrollo de las imágenes de Hyperledger fabric para que los contenedores dentro de los pods se ejecuten con un usuario diferente al root es necesario crear un Dockerfile para cada imagen.

Todas las imágenes *rootless* se encuentran en el Anexo 7.

Un ejemplo de ello es el siguiente:

```
FROM hyperledger/fabric-peer:amd64-2.0.1
RUN addgroup --gid 1022 chaincode && adduser --uid 1022 --disabled-
password --ingroup chaincode chaincode && \
    chown -R 1022:1022 /var/hyperledger/* && \
    chown -R 1022:1022 /var/run/* && \
    chmod 744 /var/hyperledger && \
    mkdir -p /opt/gopath/src/github.com/hyperledger/fabric/peer/ &&
\
    chown -R 1022:1022 /opt/* && \
    chown -R 1022:1022 /etc/hyperledger/*
    mkdir -p
USER 1022
```

En el caso de la imagen del *chaincode,* se ha podido implementar en base RedHat.

```
# This image is a microservice in golang for the Addendum chaincode
FROM registry.redhat.io/rhel8/go-toolset:1.12.8  AS build
COPY ./ /go/src/github.com/addendum
WORKDIR /go/src/github.com/addendum
# Build application
RUN scl enable go-toolset-1.12.8 'go mod init github.com/addendum'
RUN go mod init github.com/addendum
RUN scl enable go-toolset-1.12.8 'go build -o chaincode -v .'
RUN go build -o chaincode -v .
# Production ready image
# Pass the binary to the prod image
FROM registry.redhat.io/rhel8/go-toolset:1.12.8 as prod
COPY       --from=build       /go/src/github.com/addendum/chaincode
/app/chaincode
USER 1022
WORKDIR /app
CMD ./chaincode
```

- **Límite de recursos**

Los recursos en el cluster *on prermise* están limitados por *namespace*. Se ha proporcionado un *namespace* limitado para cada entorno (tst, pre y pro) que permite utilizar un determinado número de recursos. Para evitar fallo y caídas en los *deployments* por el uso descontrolado de recursos, se ha optado por introducir en los fichero de configuración yaml lo siguiente:

```
        resources:
          requests:
            cpu: 500m
            memory: 500M
          limits:
            cpu: 500m
            memory: 500M
```

Mediante esta configuración se limita al *pod* en recursos de cpu y memoria y el kube-scheduler es capaz de conocer donde desplegar y ejecutar los pods. La unidad de CPU son los milicores, mientras que la de memoria es el estándar.

Para dimensionar de forma coherente los pods de la red Hyperdledger Fabric hemos realizado una prueba de estrés a través de la API-REST. Mediante la aplicación Postman se han lanzado contra la API y, en consecuencia, contra la red blockchain una gran cantidad de peticiones. Estas peticiones se traducen en transacciones de lectura y escritura en la *blockchain*. De esta forma, se puede demostrar el máximo de recursos que consumen todos los microservicios.

En la situación de estrés, los pods de la red *blockchain* llegan al máximo de CPU y memoria que se indica en la tabla 14.

Tabla 14. Indica los recursos consumidos por los pods bajo una prueba de estrés.

POD	CPU (#Cores)	MEM	DISK	#UNITS
Peer	1	300M	100M	2
CouchDB-peer	1	300M	100M	2
CA	MIN	MIN	MIN	2
Orderer	MIN	MIN	100M	3
Chaincode	1	400M	100M	2
CLI	MIN	MIN	0	1
API	1	300M	300M	2

- **Persistencia de datos**

La persistencia de datos de los deployments está garantizada mediante los *persistent volumes* y los p*ersistent volume claims*, explicados en anteriores puntos.

Los *persistent volumes* y los *persistent volumes claims* se encuentran en los archivos Anexos.

- **Habilitar la monitorización**

En un entorno productivo, sobre todo si la herramienta consta de tantos componentes como es el caso, es necesario el pleno conocimiento del estado de los *deployments* en el cluster. Por como está configurado el cluster empresarial no tenemos acceso a los logs y al estado de los recursos si no es con herramientas corporativas que lo permiten.

Por esta razón, en cada *deployment* hay que configurar la habilitación que permite el *scraping* de logs y de las métricas.

La configuración que permite habilitar la monitorización en el cluster empresarial es: `logging.elk.stack: platform`

- **Pasar los tests de calidad**

Los tests de calidad incluyen:

- Prueba de carga: se ha realizado la prueba de carga a través del *front-end*. Esta prueba se realiza mediante condiciones normales de usabilidad.
- Prueba de estrés: se ha simulado mediante la aplicación Postman y atacando directamente contra la API-REST. Se ha realizado una prueba con 150 peticiones por segundo, tanto de lectura como de escritura.
- Prueba de estabilidad: se hacen pruebas de funcionamiento normal y se verifica que todos los componentes funcionan correctamente.
- Pruebas de pico: se han realizado sobre el frontal. Un elevado número de usuarios ha probado si la funcionalidad de la *webapp* se ve afectada.

4.4.5.1. Tests de resiliencia de la red HyperIdger Fabric

Este tipo de tests es necesario para determinar el diseño de High Availability de la red Hyperledger Fabric. Este test consiste en eliminar diferentes componentes de la red Hyperledger y ver cómo reacciona frente a estos fallos.

Con esto ganamos dos cosas, la primera nos permite conocer posibles fallos. Es decir, nos permite tener documentados los fallos por si vuelven a pasar, ya los tenemos identificados. La segunda, conocemos el límite de la red.

La red sigue funcionando si:

- Si eliminamos un *pack* chaincode-peer-couchDB. La API-REST balancea hacia el *peer* que está levantado. Cuando se vuelve a levantar este pack, si se han realizado transacciones en la red, el peer actualiza la ledger automáticamente.
- Si eliminamos un *orderer*. Si se elimina el *orderer* líder en ese momento el protocolo de ordenación elige a otro líder automáticamente.
- Si eliminamos una de las APIs, la otra sigue realizando transacciones a la red.
- Si se eliminan las dos CAs. Ya que las peticiones se hacen a través de la API que también está registrada como una organización más, pero sin serlo. Al cabo del tiempo, el funcionamiento de la red se vería interrumpido sin CAs.

La red no funciona si:

- Se eliminan solo dos *chaincodes*.
- Se eliminan los dos *couchDB*.
- Se eliminan los dos *peers*.
- Se eliminan dos o más *orderers*.

4.4.5.2. Diseño de la High Availability del sistema

La alta disponibilidad se tiene en cuenta a varios niveles:

Nivel de centro de datos: la alta disponibilidad de la infraestructura viene dada porque se encuentra repartida en dos centros de datos distintos.

Nivel kubernetes: la alta disponibilidad viene dada porque se encuentra en *workers* de diferentes clústeres.

Nivel Hyperledger Fabric: la alta disponibilidad viene dada por el número de *orderers* y de peers.

Nivel de NFS: La alta disponibilidad de los datos se mantiene debido a que cada *ledger* se almacena en NFS distinos.

Nivel de datos: gracias a lo anterior, los datos se encuentran replicados en dos NFS distintos. El NFS común almacena las credenciales de la API.

El único punto de fallo identificado es el de las credenciales de la API. Si fallara el NFS común, la API no podría funcionar.

En la siguiente figura 24 podemos ver como quedaría la infraestructura desplegada en multicentro de datos con HA.

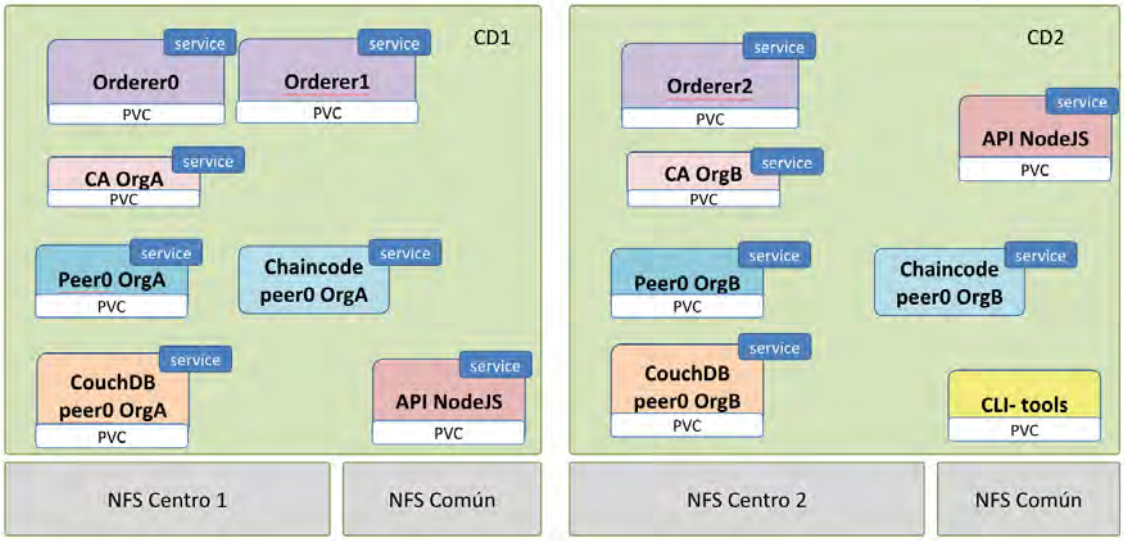


Figura 24. Despliegue en entorno de TST on premise, con Alta disponibilidad de los componentes Hyperldger Fabric.

4.5.5.3. Diseño la monitorización

La monitorización de una aplicación es necesaria para prevenir posibles fallos o, para conocer las razones de esos a posteriori. Es la única forma de saber
Por lo que es muy importante conocer qué métricas son significativas para cada componente. Las métricas del peer

La monitorización de los microservicios desplegados en el entorno se realiza a través de dos aplicaciones diferentes, una se encarga de recoger e indexar los logs para visualizarlos y la otra permite crear *dashboards* y gráficas mediante las métricas que extrae el microservicio de docker en concreto.

La aplicación que permite visualizar los logs se llama Kibana. El Kibana es una aplicación que forma parte de un conjunto llamado ELK, Elasticsearch Logkstash y Kibana.
El *Stack ELK* se encarga de recolectar los logs de un cluster, formatear y poner en común todos los *logs* que provienen de diferentes pods y, luego aporta mediante el Kibana, una *user interface.*

Por otra parte, el conjunto de aplicaciones que permiten la recolección de métricas es Prometheus y Grafana.
Prometheus es una herramienta que ofrece una serie de filtros configurables para recolectar métricas. Grafana mediante la ejecución de *rich queries* sobre las métricas recolectadas por la anterior aplicación, permite generar dashboards. Las gráficas que facilitan la interpretación y el conocimiento del estado de los pods.

En los documentos anexos se encuentra las imágenes de las herramientas de monitorización diseñadas para la aplicación. Anexo 9

4.5.5.4. Diseñar la política de Backup de los datos

Se pacta con el cliente que va a utilizar la herramienta y el departamento de infraestructura la criticidad de los datos. En función del grado de criticidad de los datos, los backups de los NFS se realizan de manera más seguida.

4.5. Despliegue de la red HyperIdger Fabric v2.0.1 on-premise

Mediante el siguiente punto se quiere describir al lector el proceso de despliegue de una red Hyperledger Fabric en objetos de kubernetes en el entorno TST *on-premise* de un *cloud* del sector bancario.
El objetivo de este punto es conseguir todos los deployments en color verde, es decir, los pods en estado *Runnig.* A continuación se muestra la siguiente imagen del *dashboard* de Kubernetes:

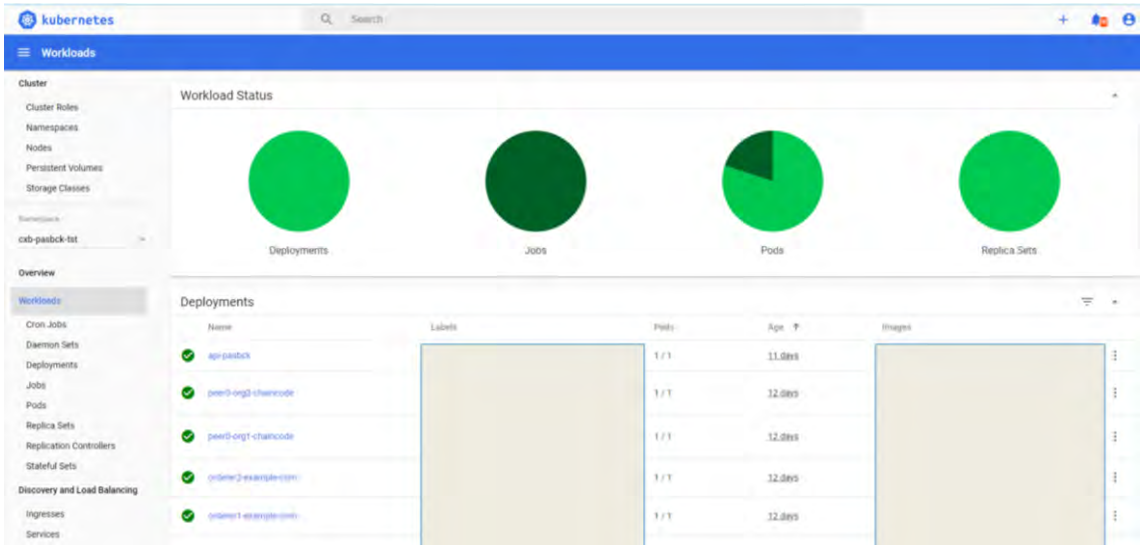


Figura 25. *Dashboard* de Kubernetes representando el estado de los pods desplegados en el el entorno tst de ITnow.

Configuraciones previas

La red que queremos desplegar es en kubernetes por lo que primero, para poder desplegar el *chaincode* como un servicio externo. Hyperledger Fabric pone a disposición una serie de configuraciones, la primera tiene que realizarse en el fichero core.yaml que sirve para configurar las variables de entorno de los *peers*.
Para hacer el despligue en modo External Builder, el peer tiene que ser configurado de la siguiente forma en el core.yaml:

```
externalBuilders:
   - name: mygolangbuilder
     path: /builders/golang/
     environmentWhitelist:
         - GOPROXY
```

El proceso external builder está basado en los Heroku buildpacks que automatizan la creación de procesos de cualquier *framework* o lenguaje. Los buildpacks de Hyperldger fabric funcionan con 3 scripts *detect*, *build i release*. Estos pueden encontrarse en el Anexo 10.

En el momento de que el peer reciba la petición “peer chaincode install chaincode-name.tar”, el proceso *external-builder* desencadena la ejecución de los *scripts*.

Por otro lado, se tiene que realizar la configuración referente al paquete en formato *tar* que lanza el comando “peer chaincode install chaincode-name.tar”.
En otras palabras, antes de empezar con el despliegue se tiene que elaborar previamente el paquete chaincode-name.tar.
Para explicar el proceso, se ha puesto de ejemplo la configuración del paquete addendum-org0alfa.tgz.

El paquete addendum-org0alfa.tgz ha sido elaborado con los siguientes archivos:

- El archivo metadata.json: incluye la información del tipo de ciclo de vida con el que se va a instalar el *chaincode*. En este caso, queremos instalar el *chaincode* como un microservicio externo al peer. Por lo que en el archivo *json,* es necesario poner `"type":"external".` El *`path`* es donde se va a cagar el directorio en el que se encuentra el *chaincode,* *`label`* es el nombre y el versionado que le proporcionaremos al *chaincode*

  `{"path":"","type":"external","label":"addendum_v1"}`

- El archivo connection.json presenta la información de la comunicación que se va crear entre peer y chaincode. En este fichero hay que rellenar el campo `"address"` con el FQDN que va a tener el *chaincode* en el cluster de kubernetes. La dirección incluye el puerto por el que escucha el servicio del *pod* del *chaincode*, el 7052.

```
{"address":"peer0-org0beta-chaincode:7052",
"dial_timeout": "10s",
 "tls_required": false,
"client_auth_required": false,
"client_key": "-----BEGIN EC PRIVATE KEY-- ... --END EC PRIVATE KEY-----",
"client_cert": "-----BEGIN CERTIFICATE----- ... -----END CERTIFICATE-----",
"root_cert": "-----BEGIN CERTIFICATE---- ... -----END CERTIFICATE-----"}
```

Una vez configurados estos dos archivos *json* se disponen en este sistema de directorios y se empaquetan concurrentemente siguiendo estos pasos:

```
path-peer/chaincodes/
            /connection.json
            /metadata/metadata.json
            /metadata.json
```

Primero se genera el paquete code.tar.gz mediante el siguiente comando de empaquetado. Con este comando generamos el paquete code.tar.gz que contiene el fichero connection.json , el directorio metadata con el metadata.json: `/metadata/metadata.json`

```
tar cfz code.tar.gz connection.json metadata
```

Ahora el directorio, con el nuevo paquete generado queda:

```
path-peer/chaincodes/
            /code.tar.gz
            /connection.json
            /metadata/metadata.json
            /metadata.json
```

Por último, se coge la copia del fichero metadata.json junto con el paquete generado code.tar.gz y se genera el último fichero comprimido:

```
tar cfz addendum-org0alfa.tgz metadata.json code.tar.gz
```

El directorio queda finalmente de esta forma:

```
path-peer/chaincodes/
        /addendum-org0alfa.tgz
        /code.tar.gz
        /connection.json
        /metadata/metadata.json
        /metadata.json
```

Este proceso de empaquetado se ha realizado para que el proceso External Builders tenga la información necesaria para crear la imagen del chaincode. El paquete addendum-org0alfa.tgz se almacena en el directorio *chaincodes* de los *Peers*. Para concluir el paquete addendum-org0alfa.tgz contiene:

```
/addendum-org0alfa.tgz
                    /metadata.json
                    / code.tar.gz
```

A su vez, code.tar.gz presenta:

```
/code.tar.gz
        /connection.json
        /metadata/metadata.json
```

Instalación y despliegue

Primero de todo, el despliegue se va a realizar mediante *jobs* de kubernetes. Los *jobs* son *pods* que se crean con un propósito en concreto, al realizar esa tarea se eliminan automáticamente del cluster liberando los recursos. La naturaleza de los *jobs* los hace perfectos para despliegues de este tipo.

Por otro lado, está totalmente prohibido lanzar comandos manuales al cluster corporativo. Existe un equipo encargado de supervisar el código que se despliega, que está durante el despliegue de los componentes, monitorizando los *deployments* que se ejecutan y supervisando a los desarrolladores para que no se realice ningún tipo error.

El hecho de que se realice la instalación mediante jobs es perfecto para la configuración de la red *blockchain*. Los *jobs* van a tener la imagen del CLI de Hyperledger Fabric. Esta imagen contiene todos los binarios y herramientas necesarias, como el *cryptogen* y el *configtx*, la configuración de la *Blockchain*.

Los archivos que definen la red *blockchain* son el configtx.yaml y el crypto-config.yaml. La forma de definir la topología de la red Hyperledger Fabric es mediante el archivo configtx.yaml. Incluye los participantes de la red, los *peers* de cada participante, la definición del servicio *ordering*, el control de acceso al canal (ACL) y las políticas de aprobación de las transacciones.
El archivo crypto-config.yaml está estructurado en dos partes. La primera contiene la definición de los nombres de cada nodo que forma parte del servicio de ordering. La segunda presenta la definición de los nombres de cada *peer* que va a formar parte de cada organización en la red. El archivo crypto-config.yaml permite generar los certificados y claves para cada organización. Este crea una *Certificate Authority* para cada organización y también se encarga de crear las credenciales que va a utilizar el componente abstracto MSP.

El despliegue se tiene que realizar siguiendo este orden:

1. cli_job1_config_files.yaml
    a. ca0A
    b. ca0B
    c. couchDB0A
    d. peer0A
    e. couchDB0B
    f. peer0B
    g. orderer0
    h. orderer1
    i. orderer2
2. cli_job02_channel_chaincode.yaml
    a. chaincode0A
    b. chaincode0B
3. cli_job03_approve_commit_init.yaml

El primer job se despliega, genera un *pod* que lanza el *script :* `01_steps1-5_generate-bckfiles.sh`

El *script* se divide en cinco pasos:

El primer paso genera el material criptográfico invocando la herramienta *crypto-config.* Esta consume el fichero crypto-config.yaml. Al finalizar la herramienta genera dos directorios donde almacena los certificados y claves para las organizaciones.

```
STEP 1: Cryptogen Starts
org0alfa-pasbck-new-com
org0beta-pasbck-new-com
```

El segundo, tercer, cuarto y quinto paso genera la configuración del canal invocando la herramienta *configtxgen*. Esta consume el fichero configtx.yaml. Al finalizar la herramienta genera 4 ficheros:

- genesis.block: Este fichero representa el primer bloque de la *blockchain*, el genesis block. El bloque será entregado al servicio de *ordering* cuando este se despliegue.
- channel.tx: Este fichero contiene la configuración del canal de la red Hyperledger Fabric.
- Org0alfaMSPanchors.tx y Org0betaMSPanchors.tx: Estos dos ficheros contienen la configuración de los *peers* que son tipo Anchor.

El objetivo que se quiere conseguir con este primer *job* es generar los archivos de configuración que luego serán consumidos por los componentes de Hyperledger Fabric. Con este primer despliegue damos por concluida el *setup* de la red.

El resultado del despliegue del primer job cli_job1_config_files.yaml se ve en la siguiente imagen:

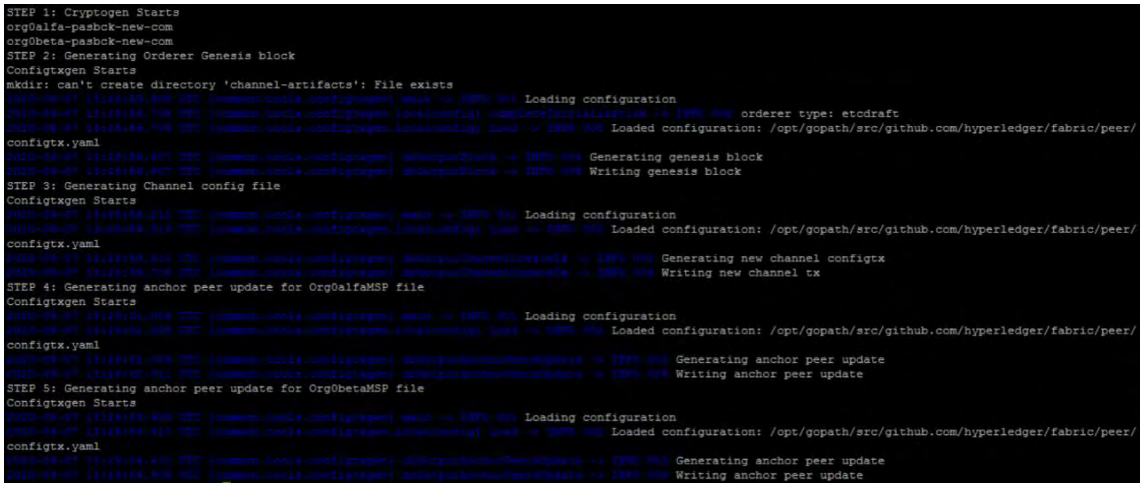


Figura 26. Resultado de la ejecución de la herramienta configtxgen y crypto-config.

Seguidamente, se procede a desplegar los componentes de la red Hyperledger Fabric. Primero el despliegue de las CAs que usarán los directorios creados en el primer paso del *script*. Las CAs acceden al sistema de ficheros con las credenciales para dotar a los participantes de la red el *rootCertificate* y el *enrollment Certificate*.

Los despliegues del *peer* y el couchDB se recomienda hacerlos seguidos en tiempo. Ya que, una vez se levanta el couchDB empieza a escuchar por el puerto 5984 y si no encuentra el peer en ese momento deja de buscarlo. Pasado unos segundos vuelve a intentar encontrar el *peer*, cada vez estos intervalos son más largos. Si el couchDB no se despliega con el *peer* este al cabo de un dia, entra en estado de fallo *CrashLoopBackOff*.

A continuación, se despliegan los orderers, que una vez están los tres *pods* en estado *Running* el servicio de *ordering* se pone en marcha y activa el protocolo de consenso RAFT. Los *orderers* empiezan las comunicaciones entre ellos para elegir quien será el líder y quien los *followers.* El servicio de *ordering* tiene acceso al fichero genesis.block y el servicio envia en formato de bloque genesis a los *peers*. Estos guardan el bloque en la ledger mantenida por los couchDB.

El siguiente job cli_job02_channel_chaincode.yaml, lanza los *scripts:* `02_steps6-7_create_join_channel_alfa.sh`

`03_steps8-9_create_join_channel_beta.sh`

`04_steps10a_channel_list.sh`

`05_steps10b_channel_list.sh`

`06_steps11_chaincodealfa_install.sh`

`07_steps12_chaincodebeta_install.sh`

El *script* `02_steps6-7_create_join_channel_alfa.sh` contiene en su interior los comandos para que el peerA0 cree el canal y se una a él.

El *pod* del CLI cuando ejecuta este *script* apunta al peer0A y lanza los comandos peer channel create y peer channel join.

Peer channel create utiliza el fichero de configuración del canal `channel.tx` para crear por primera vez el canal.

```
peer channel create -o orderera-pasbck-new-com:7050 -c $CHANNEL_NAME -f ./channel.tx --tls true --cafile $ORDERER_CA
```

Cuando finaliza este comando se crea el bloque que contiene la configuración del canal (channel.block)

"Peer channel join" permite al Peer0A unirse al canal mediante el channel.block.

```
peer channel join -b $CHANNEL_NAME.block
```

El resultado del script "02_steps6-7_create_join_channel_alfa.sh" es el siguiente:

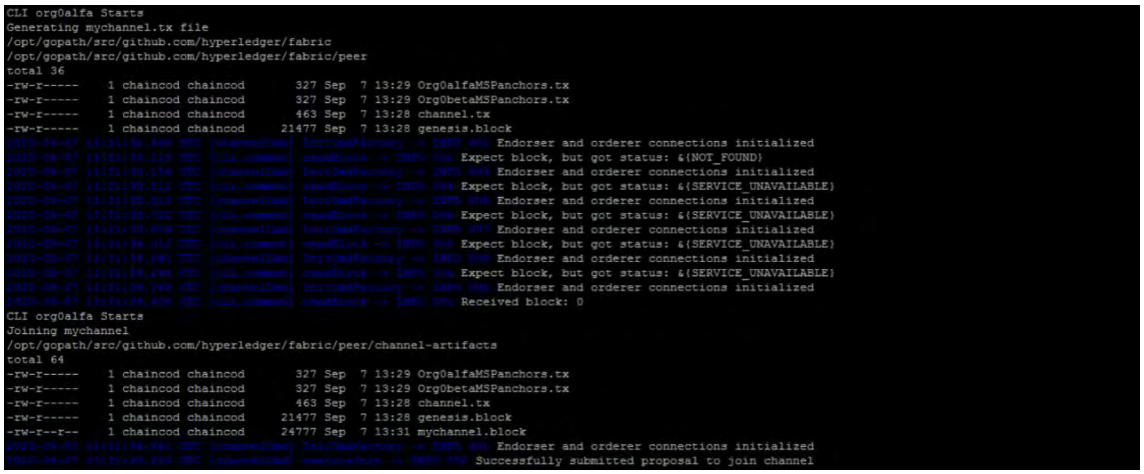


Figura 27 . Resultado de la ejecución de los comandos de crear y unir al canal por parte del Peer0A.

El *script* `03_steps8-9_create_join_channel_beta.sh` ejecuta exactamente los dos mimos comandos pero apuntando al el peer0B.

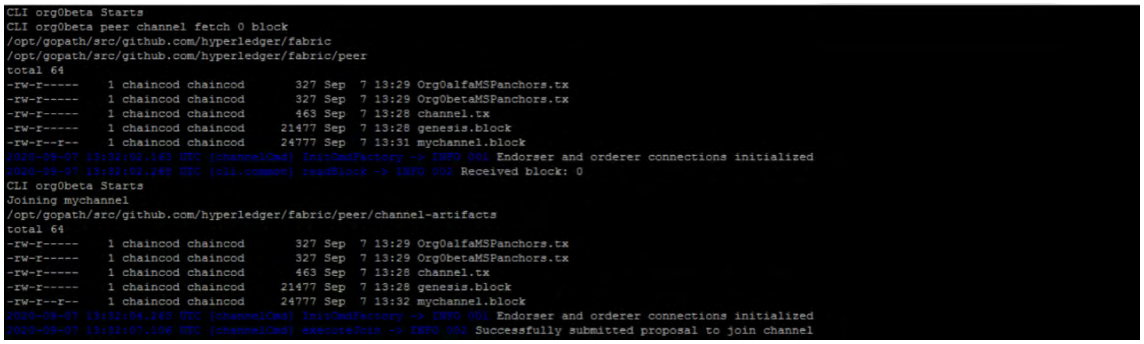


Figura 28. Resultado de la ejecución de los comandos de crear y unir al canal por parte del Peer0B.

El *script* `04_steps10a_channel_list.sh` hace que el *pod* de CLI lance el comando "peer channel list" apuntando al Peer0A a modo de comprobación. Con este comando se comprueba si por parte del Peer0A se ha creado el canal y se ha unido a él.

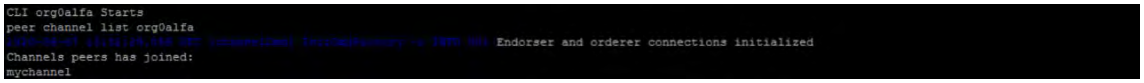


Figura 29. Resultado de la ejecución de la query lanzada al Peer0A para comprobar que se ha creado el canal y se ha unido a él. El Peer0A lista los canales a los cuales está unido.

El *script* `05_steps10b_channel_list.sh` hace lo mismo que el script anterior pero apuntando al Peer0B. En este caso el Peer0B responde al comando "peer channel list", lo siguiente:

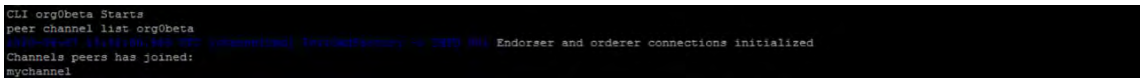


Figura 30. Resultado de la ejecución de la query lanzada al Peer0B para comprobar que se ha creado el canal y se ha unido a él. El Peer0B lista los canales a los cuales está unido.

El *script* `06_steps11_chaincodealfa_install.sh` hace que el pod del CLI apunte al Peer0A y lanza el comando peer lifecycle chaincode install addendum-org0alfa.tgz .

El Peer0A analiza el comando e identifica que se trata de la instalación del chaincode. En este preciso momento, el peer ejecuta el proceso external-builder desencadenando la ejecución de los *scripts: detect, build y release.*

- El *script* ***detect*** simplemente evalúa si el chaincode que se está instalando tiene la clave `"type":"external"` en el fichero metadata.json. Si este *script* falla, el peer considera que el chaincode no tiene que instalarse siguiendo el proceso external builder y creará la imagen del chaincode mediante el docker daemon. Es decir, se inicia el proceso por defecto de instalación del chaincode.

  Una vez finaliza correctamente el *script detect,* se llama automáticamente al *script build*.

- El *script* ***build*** copia la configuración del fichero connection.json, procesa los datos que hay en su interior y los muestra ordenados en el *output script.*
- Por último, se llama al *script* ***release.*** Este *script* coge el output del *build* y transmite esa información al peer. Por lo tanto, el peer ahora conoce en qué directorio y de qué forma tiene que llamar al chaincode si necesita invocarlo.

La ejecución del proceso External-Builders en los peers produce la siguiente salida:

```
Chaincode code package identifier:
addendum_v1:1528162346496c7dec8637bf769a1f365e84dc14b38ed4f27363ca41db3ea37d
```

El chaincode package identifier (CCID) es el identificador que le otorga el peer al chaincode. Este CCID es representativo de la configuración del *chaincode,* si el *chaincode* es cambiado, no se corresponde con el CCID que tiene guardado el peer.

El *script* `07_steps12_chaincodebeta_install.sh` hace lo mismo que el anterior pero para el Peer0B.

La salida de estos dos últimos scripts está en la siguiente figura:

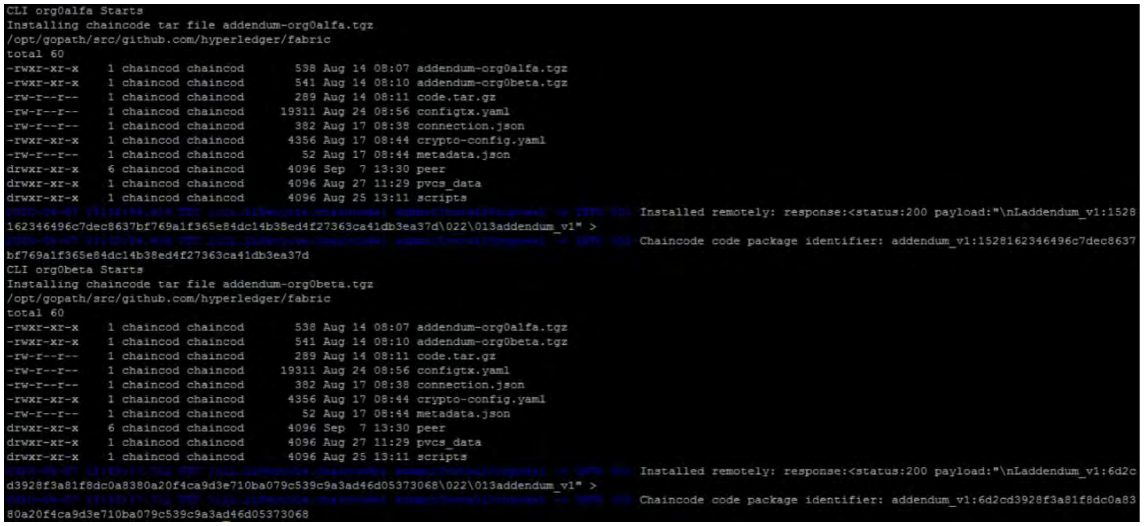


Figura 31. Representa los CCID del Peer0A y el Peer0B.

En este punto se detiene el despliegue de *jobs* de kubernetes para editar los ficheros chaincode0alfa.yaml y chaincode0beta.yaml. Estos son los ficheros de configuración del *deployment* de los *chaincodes*.

La configuración a realizar es la siguiente, hay que copiar el CCID generado del comando "peer chaincode install addendum-org0alfa.tgz", en las variables de entorno del *deployment* del *chaincode*.

```
containers:
  - image: chaincode/redhat-chaincode:2
    name: peer0-org0alfa-chaincode
    imagePullPolicy: IfNotPresent
    env:
```

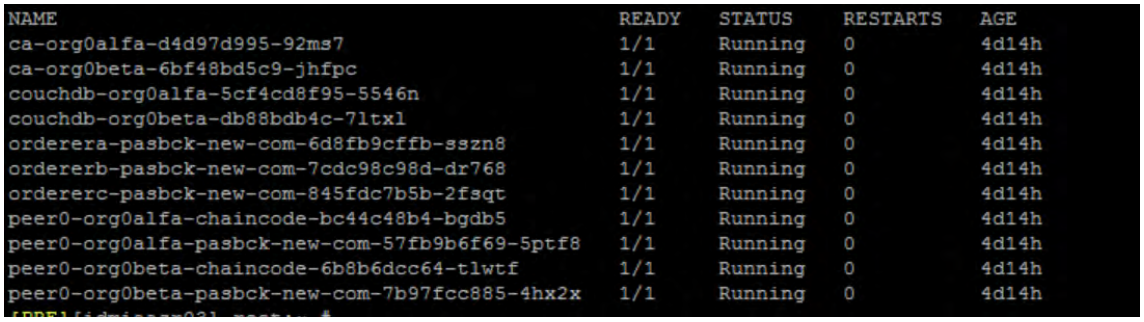
```
            - name: CHAINCODE_CCID
        value:
"addendum_v1:1528162346496c7dec8637bf769a1f365e84dc14b38ed4f27363ca41db3ea37d"
```

De esta forma, el *chaincode* queda configurado con el CCID que le ha proporcionado el *peer*.

Cuando los *chaincodes* se despliegen, y los *peers* inicien las conexiones con el *chaincode*, comprobarán antes de conectarse si los CCID concuerdan.

Al finalizar las dos configuraciones de los ficheros yaml de los *chaincodes*, se procede al despligue de los deployments: chaincode0A y chaincode0B.

Con estos dos despliegues, se tienen todos los componentes de la red Hyperdleger Fabric levantados y funcionando en el cluster de kubernetes:



NAME	READY	STATUS	RESTARTS	AGE
ca-org0alfa-d4d97d995-92ms7	1/1	Running	0	4d14h
ca-org0beta-6bf48bd5c9-jhfpc	1/1	Running	0	4d14h
couchdb-org0alfa-5cf4cd8f95-5546n	1/1	Running	0	4d14h
couchdb-org0beta-db88bdb4c-7ltxl	1/1	Running	0	4d14h
orderera-pasbck-new-com-6d8fb9cffb-sszn8	1/1	Running	0	4d14h
ordererb-pasbck-new-com-7cdc98c98d-dr768	1/1	Running	0	4d14h
ordererc-pasbck-new-com-845fdc7b5b-2fsqt	1/1	Running	0	4d14h
peer0-org0alfa-chaincode-bc44c48b4-bgdb5	1/1	Running	0	4d14h
peer0-org0alfa-pasbck-new-com-57fb9b6f69-5ptf8	1/1	Running	0	4d14h
peer0-org0beta-chaincode-6b8b6dcc64-tlwtf	1/1	Running	0	4d14h
peer0-org0beta-pasbck-new-com-7b97fcc885-4hx2x	1/1	Running	0	4d14h

Figura 32. Resultado del despliegue de todos los componentes de hyperladder fabric.

Se han desplegado todos los componentes de Hyperldger fabric, pero aún falta acabar de configurar la red. Para ello se despliega el siguiente *job:* cli_job03_approve_commit_init.yaml

Este lanza los *scripts* :

`08_steps13_approve_alfa.sh`

`09_steps14_approve_beta.sh`

`10_steps15_checkcommitreadiness_alfa.sh`

`11_steps16_checkcommitreadiness_beta.sh`

`12_steps17-19_channel_chaincode.sh`

Los *scripts* `08_steps13_approve_alfa.sh` y `09_steps14_approve_beta.sh` lanzan los comandos de aprobación del *chaincode*. Las organizaciones participan en un mismo canal, por lo tanto las dos tienen el mismo chaincode. Pero previamente a instalarlo

en su infraestructura, pueden revisarlo y decidir si lo aprueban o no. Esto se hace mediante el comando" peer lifecycle chaincode approveformyorg". Este proporciona los parámetros de definición del *chaincode* como la gobernanza del *chaincode*, el nombre y la versión de este. También se incluye el parámetro del CCID asociado al *chaincode package* instalado en los *peers* de la organización.

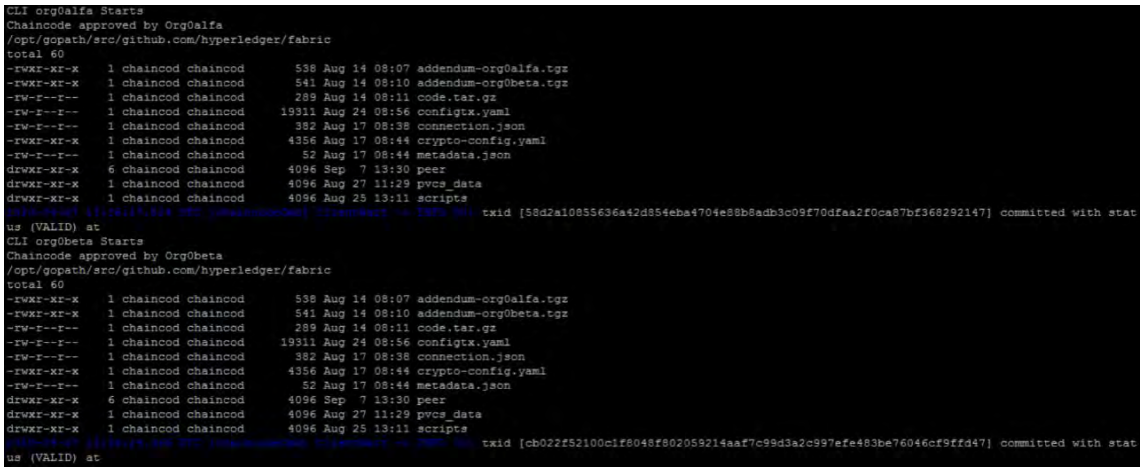


Figura 33. Resultado de la aprobación del *chaincode* por parte de las Organizaciones.

Cuando el *chaincode* es aprobado por las dos organizaciones se continua con los siguietes dos *scripts* `10_steps15_checkcommitreadiness_alfa.sh` y

`11_steps16_checkcommitreadiness_beta.sh`

Estos dos scripts lanzan el comando "peer lifecycle chaincode checkcommitreadiness" a modo de *query* para saber si las organizaciones han aprobado correctamente el *chaincode*.

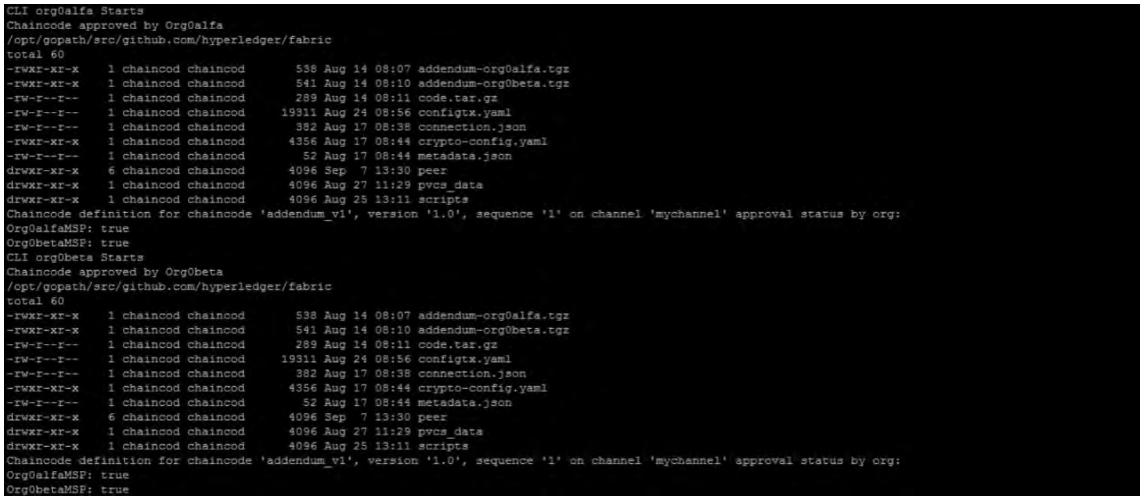


Figura 34. Resultado de la comprobación de aprobación del *chaincode* por parte de las organizaciones

El resultado de los comandos es un fichero *json* con las organizaciones que forman parte del canal, diciendo si han aprobado o no el *chaincode.*

```
{
        "Approvals": {
                "Org0alphaMSP": true,
                "Org0betaMSP": true
        }
}
```

Finalmente, el utlimo *job* lanza el *script* `12_steps17-19_channel_chaincode.sh`. Después de que el *chaincode* se haya aprobado en el canal, los peers están listos para hacer la primera invocación del *chaincode* y empezar a interactuar con la *ledger.*

El *script* contiene el comando "peer chaincode invoke" que realizará el primer *invok*e al *chaincode* de la organinzació Alfa. Para ello se necesita pasar a este comando el parámetro `--IsInit`. Este parámetro llama al método Init del *chaincode* y hace que se ejecute.

El comando es el siguiente:

```
peer chaincode invoke -o orderera-pasbck-new-com:7050 --channelID
mychannel --isInit --name addendum_v1 --tls true --cafile
$ORDERER_CA --peerAddresses peer0-org0alfa-pasbck-new-com:7051 --
tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto-
config/peerOrganizations/org0alfa-pasbck-new-com/peers/peer0-
org0alfa-pasbck-new-com/tls/ca.crt --peerAddresses peer0-org0beta-
pasbck-new-com:7051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto-
config/peerOrganizations/org0beta-pasbck-new-com/peers/peer0-
org0beta-pasbck-new-com/tls/ca.crt -c '{"Args":["Init"]}' --
waitForEvent
```

El primer *invoke* al *chaincode* ejecuta el contendor del *chaincode* dentro del deployment de los *chaincodes*. Este proceso dura unos minutos.

Por último lanzamos un invoke a un método del *chaincode*, por ejemplo en nuestro caso, el método que lista las adendas guardadas en la blockchain. Este invoke lanza una propuesta de transacción que tiene que ser ejecutada por todas las organizaciones, por lo que el *chaincode* de la organización Beta, tendrá que levantar el contendor del chaincode y ejecutar la propuesta de tranacción.

```
peer chaincode invoke -o orderera-pasbck-new-com:7050 --channelID
mychannel --name addendum_v1 --tls true --cafile $ORDERER_CA --
peerAddresses peer0-org0alfa-pasbck-new-com:7051 --
tlsRootCertFiles
```

```
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto-
config/peerOrganizations/org0alfa-pasbck-new-com/peers/peer0-
org0alfa-pasbck-new-com/tls/ca.crt --peerAddresses peer0-org0beta-
pasbck-new-com:7051 --tlsRootCertFiles
/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto-
config/peerOrganizations/org0beta-pasbck-new-com/peers/peer0-
org0beta-pasbck-new-com/tls/ca.crt -c
'{"Args":["getAllAddendums"]}' --waitForEvent
```

Finalmente, ha finalizado la instalación de la red Hyperldger Fabric en el entorno de kuernetes.

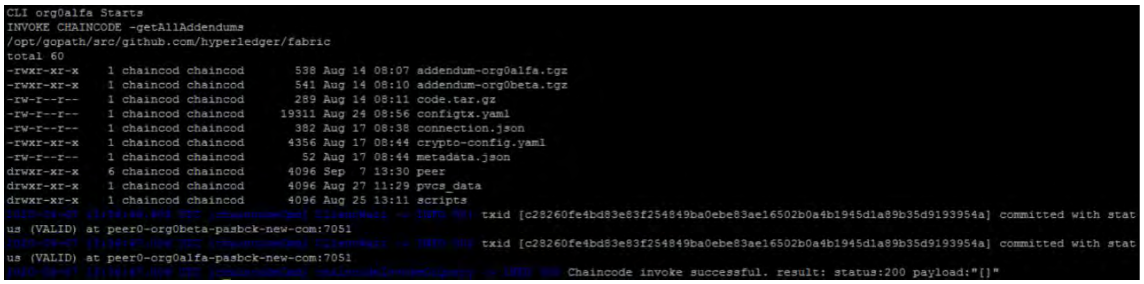


Figura 35. Invocación de chaincode satisfactoria

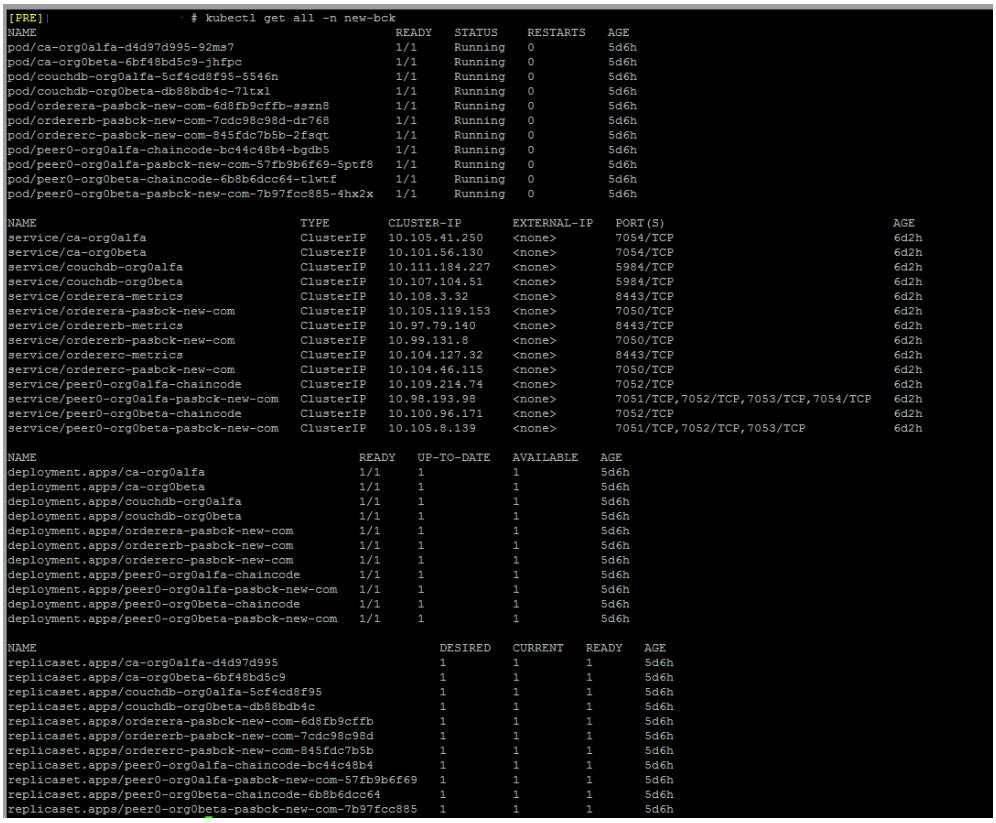
Los archivos de configuración crypto-config y configtx se encuentran en el anexo 8, mientras que los archivos de configuración de los componentes de Hyperldger Fabric se encuentran en el Anexo 5.

Capítulo 5: Resultados

Se ha podido comprobar que el funcionamiento y la respuesta de la *blockchain* es más que correcta. La herramienta de adendas es capaz de guardar y obtener información de la *blockchain*, de proveer acceso a ella y de determinar las condiciones que deben cumplirse para poder escribir en la *blockchain*, respetando el flujo impuesto por el documento comercial de las adendas.

En esta sección se detallan los principales resultados, aunque no se especificarán todos y cada uno de ellos para no dilatar de manera excesiva el capítulo.

Tabla 15 . Caso de prueba 1

Descripción	Despliegue de todos los componentes de la Hyperledger Fabric en el cluster de kubernetes corporativo.
Resultado obtenido	Despliegue de todos los contenedores en correcto estado, la red correctamente configurada y funcionando de forma coherente con el caso de uso.
Evidencia	
Conclusión	Satisfactoria

```
[PRE]                    # kubectl get all -n new-bck
NAME                                                READY   STATUS    RESTARTS   AGE
pod/ca-org0alfa-d4d97d995-92ms7                     1/1     Running   0          5d6h
pod/ca-org0beta-6bf48bd5c9-jhfpc                    1/1     Running   0          5d6h
pod/couchdb-org0alfa-5cf4cd8f95-5546n               1/1     Running   0          5d6h
pod/couchdb-org0beta-db88bdb4c-7ltxl                1/1     Running   0          5d6h
pod/orderera-pasbck-new-com-6d8fb9cffb-sszn8        1/1     Running   0          5d6h
pod/ordererb-pasbck-new-com-7cdc98c98d-dr768        1/1     Running   0          5d6h
pod/ordererc-pasbck-new-com-845fdc7b5b-2fsqt        1/1     Running   0          5d6h
pod/peer0-org0alfa-chaincode-bc44c48b4-bgdb5        1/1     Running   0          5d6h
pod/peer0-org0alfa-pasbck-new-com-57fb9b6f69-5ptf8  1/1     Running   0          5d6h
pod/peer0-org0beta-chaincode-6b8b6dcc64-tlwtf       1/1     Running   0          5d6h
pod/peer0-org0beta-pasbck-new-com-7b97fcc885-4hx2x  1/1     Running   0          5d6h

NAME                                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                          AGE
service/ca-org0alfa                     ClusterIP   10.105.41.250    <none>        7054/TCP                         6d2h
service/ca-org0beta                     ClusterIP   10.101.56.130    <none>        7054/TCP                         6d2h
service/couchdb-org0alfa                ClusterIP   10.111.184.227   <none>        5984/TCP                         6d2h
service/couchdb-org0beta                ClusterIP   10.107.104.51    <none>        5984/TCP                         6d2h
service/orderera-metrics                ClusterIP   10.108.3.32      <none>        8443/TCP                         6d2h
service/orderera-pasbck-new-com         ClusterIP   10.105.119.153   <none>        7050/TCP                         6d2h
service/ordererb-metrics                ClusterIP   10.97.79.140     <none>        8443/TCP                         6d2h
service/ordererb-pasbck-new-com         ClusterIP   10.99.131.8      <none>        7050/TCP                         6d2h
service/ordererc-metrics                ClusterIP   10.104.127.32    <none>        8443/TCP                         6d2h
service/ordererc-pasbck-new-com         ClusterIP   10.104.46.115    <none>        7050/TCP                         6d2h
service/peer0-org0alfa-chaincode        ClusterIP   10.109.214.74    <none>        7052/TCP                         6d2h
service/peer0-org0alfa-pasbck-new-com   ClusterIP   10.98.193.98     <none>        7051/TCP,7052/TCP,7053/TCP,7054/TCP   6d2h
service/peer0-org0beta-chaincode        ClusterIP   10.100.96.171    <none>        7052/TCP                         6d2h
service/peer0-org0beta-pasbck-new-com   ClusterIP   10.105.8.139     <none>        7051/TCP,7052/TCP,7053/TCP       6d2h

NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ca-org0alfa             1/1     1            1           5d6h
deployment.apps/ca-org0beta             1/1     1            1           5d6h
deployment.apps/couchdb-org0alfa        1/1     1            1           5d6h
deployment.apps/couchdb-org0beta        1/1     1            1           5d6h
deployment.apps/orderera-pasbck-new-com 1/1     1            1           5d6h
deployment.apps/ordererb-pasbck-new-com 1/1     1            1           5d6h
deployment.apps/ordererc-pasbck-new-com 1/1     1            1           5d6h
deployment.apps/peer0-org0alfa-chaincode 1/1    1            1           5d6h
deployment.apps/peer0-org0alfa-pasbck-new-com 1/1 1          1           5d6h
deployment.apps/peer0-org0beta-chaincode 1/1    1            1           5d6h
deployment.apps/peer0-org0beta-pasbck-new-com 1/1 1          1           5d6h

NAME                                               DESIRED   CURRENT   READY   AGE
replicaset.apps/ca-org0alfa-d4d97d995              1         1         1       5d6h
replicaset.apps/ca-org0beta-6bf48bd5c9             1         1         1       5d6h
replicaset.apps/couchdb-org0alfa-5cf4cd8f95        1         1         1       5d6h
replicaset.apps/couchdb-org0beta-db88bdb4c         1         1         1       5d6h
replicaset.apps/orderera-pasbck-new-com-6d8fb9cffb 1         1         1       5d6h
replicaset.apps/ordererb-pasbck-new-com-7cdc98c98d 1         1         1       5d6h
replicaset.apps/ordererc-pasbck-new-com-845fdc7b5b 1         1         1       5d6h
replicaset.apps/peer0-org0alfa-chaincode-bc44c48b4 1         1         1       5d6h
replicaset.apps/peer0-org0alfa-pasbck-new-com-57fb9b6f69 1   1         1       5d6h
replicaset.apps/peer0-org0beta-chaincode-6b8b6dcc64 1        1         1       5d6h
replicaset.apps/peer0-org0beta-pasbck-new-com-7b97fcc885 1   1         1       5d6h
```

Tabla 16. Caso de prueba 2.

Descripción	Desligue del *chaincode* mediante el proceso *External-Builders* en el entorno de kubernetes corporativo.
Resultado obtenido	El contendor del *chaincode* corriendo dentro del *pod* del *deployment* del *chaincode*
Evidencia	
Conclusión	Satisfactoria

Tabla 17. Caso de prueba 3.

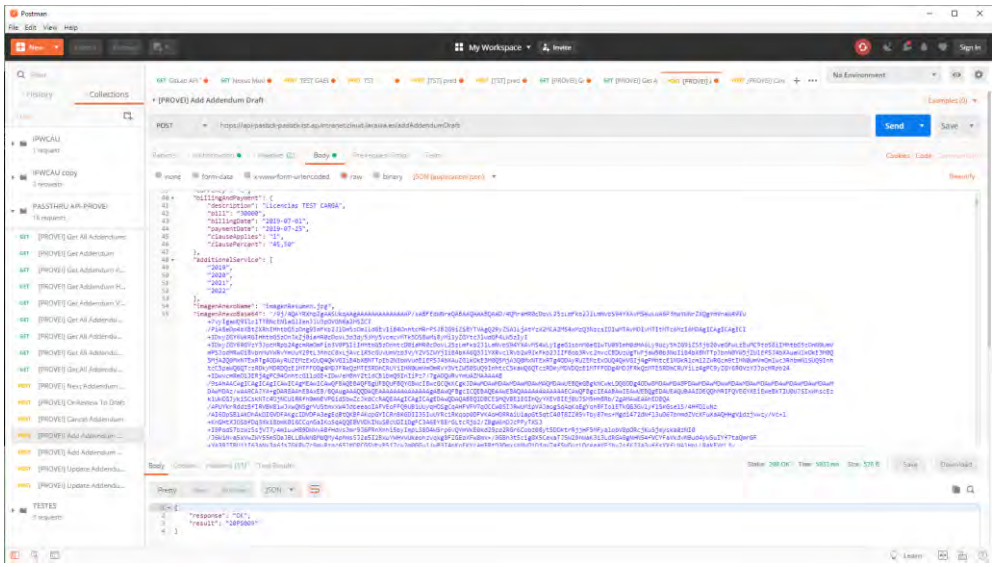
Descripción	Guardar documentos in-chain y off-chain del proceso comercial adendas en el backend basado en Hyperldger Fabric.
Resultado obtenido	Capacidad de almacenar y obtener los datos guardados en el backend. La factura ha sido gestionada por el backend y almacenada en la base de datos externa MySQL
Evidencia	
Conclusión	Satisfactoria

Tabla 18. Caso de prueba 4.

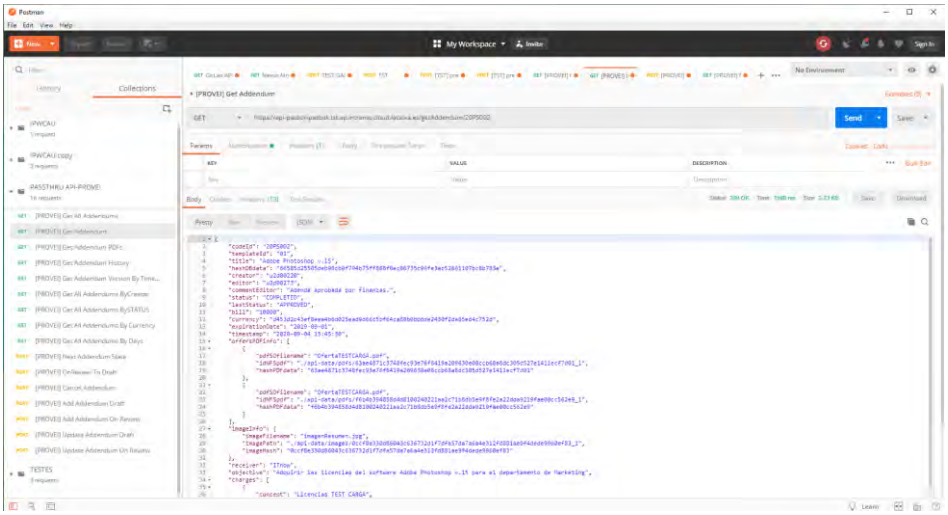
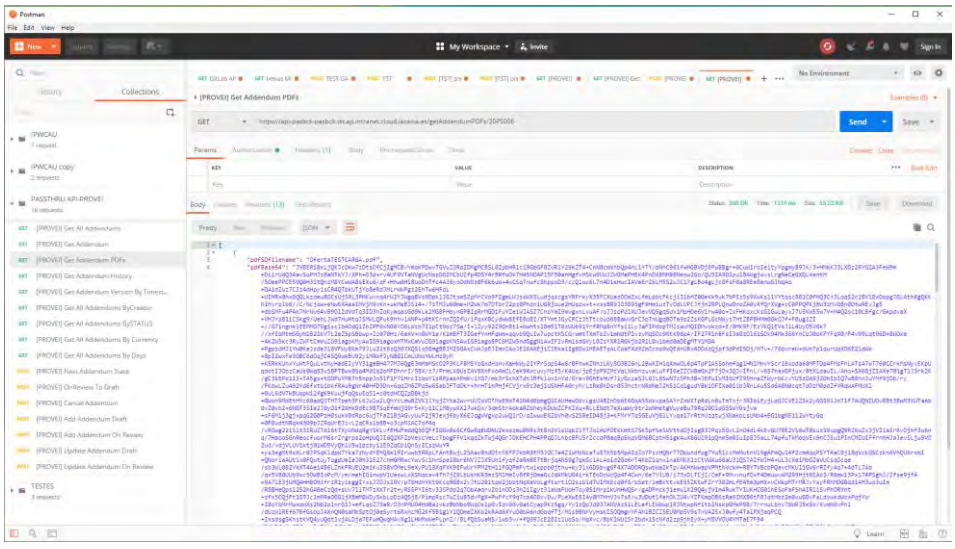
Descripción	Obtener documentos in-chain y off-chain del proceso comercial adendas proporcionado por el backend basado en Hyperldger Fabric.
Resultado obtenido	La factura ha sido proporcionada al front-end .
Evidencia	
Conclusión	Satisfactoria

Tabla 19. Caso de prueba 5.

Descripción	Obtener el documento tipo factura off-chain del proceso comercial addenas.
Resultado obtenido	La factura ha sido proporcionada al front-end .
Evidencia	
Conclusión	Satisfactoria

Capítulo 6: Conclusiones del Proyecto

El desarrollo del presente proyecto ha permitido conocer en profundidad la tecnología blockchain, sus características, las ventajas más destacadas y las implementaciones que se están llevando a cabo en múltiples sectores. Se ha estudiado especialmente el framework de Hyperdledger Fabric que ha permitido adquirir el conocimiento necesario para diseñar e implementar una *blockchain* empresarial utilizando la red Hyperledger Fabric.

La herramienta diseñada en el proyecto dada su arquitectura ha permitido el aprendizaje de almacenar datos, mantener la coherencia, la trazabilidad y la inmutabilidad de datos *off-chain*.

Con el estudio realizado queda presente el potencial para optimizar procesos legales y comerciales para empresas, pero también se ve reflejado la poca adaptabilidad y escalabilidad que ofrecen estos sistemas.

La prueba de concepto del Proyecto ha demostrado la inmadurez de los productos *open source* para ser desplegados y considerados como servicios productivos. En particular, se ha mostrado la falta de desarrollo por parte del proyecto Hyperledger Fabric, aunque su código sea en formato microservicios *cloud* no está preparado para ser orquestado mediante kubernetes.

A nivel tecnológico se han obtenido los conocimientos para desarrollar y gestionar un proyecto desde cero dentro del sector bancario. Se ha podido demostrar el contraste entre el desarrollo de una prueba de concepto y la solución productiva. El diseño y la gestión de la herramienta productiva tiene que cumplir un elevado número de requisitos si quiere formar parte de catalogo de servicios de una empresa como CaixaBank.

Por otro lado, se ha demostrado que la innovación en el sector bancario está limitada por las fuertes regulaciones legales y de seguridad. Esto provoca que muchas pruebas de concepto no puedan ser llevadas a producción.

Finalmente, se ve reflejado la gran apuesta por el Cloud en los sectores financieros para agilizar y hacer eficiente su negocio.

Capítulo 7: Líneas de Trabajo Futuras

A partir del presente Trabajo de Fin de Máster , se sugieren las siguientes propuestas de *next steps* en la tecnología *blockchain*.

- *Blockchain as a Service* como producto que cumpla las restricciones de un cloud privado.
- Infraestructura o aplicación para la auditoría y registro de los chaincodes utilizados en una red Hyperledger Fabric.
- Probar, aunque está en estado de prueba a día de hoy, los *tokens* de Hyperledger Fabric. Los *Fabtokens*.
- Investigar sobre posibles implementaciones en blockchain que fueran *GDPR Compliance approved*. Mediante mecanismos *zero-knowledge-proof* .
- Identidad Digital Soberana basada en *blockchains* públicas.

Referencias

[1] Preukschat, Alex, and Carlos Kuchkovsky. *Blockchain: La Revolución Industrial De Internet*. Booket, 2019.

[2] "Tecnologías Disruptivas En El Sector Bancario." *El Blog De CaixaBank*, 20 July 2017, blog.caixabank.es/blogcaixabank/2017/04/como-las-tecnologias-disruptivas-estan-transformando-la-banca.html.

[3] Arancibia, POR Salvador, et al. "Fain." *Expansi*, 17 Mar. 2016, www.expansion.com/empresas/banca/2016/03/17/56eb0eb6e2704e1f2e8b456b.html.

[4] *Página web corporativa ITnow*, www.itnow.es/.

[5] Sherman, Alan T., et al. "On the Origins and Variations of Blockchain Technologies." *IEEE Security & Privacy*, vol. 17, no. 1, 2019, pp. 72–77., doi:10.1109/msec.2019.2893730.

[6] Chaum, D. L. *Computer Systems Established, Maintained and Trusted by Mutually Suspicious Groups*. Electronics Research Laboratory, Univ. of California, 1979.

[7] bitcoin "Open Source P2P Money." *Bitcoin*, bitcoin.org/.

[8] *IOTA Tangle Explorer and Statistics - TheTangle.org*, thetangle.org/.

[9] "Prove Actions for Data Compliance." *Hedera Hashgraph*, 13 Sept. 2020, www.hedera.com/use-cases/data-compliance.

[10] Pattnaik, Deeptiman. "What's inside the Block in Hyperledger Fabric?" *Medium*, Medium, 19 Jan. 2020, medium.com/@deeptiman/whats-inside-the-block-in-hyperledger-fabric-69a0934fef08.

[11] Wengo W, D.T. Hoang, et al. *"*A Survey on Consensus Mechanisms and MiningStrategy Management in Blockchain Networks*" IEEE Xplore Full-Text PDF:* ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8629877.

[12] Szabo, Nick. "Formalizing and Securing Relationships on Public Networks." *First Monday*, vol. 2, no. 9, 1997, doi:10.5210/fm.v2i9.548.

[13] European Commission. "Study on Blockchains - Legal, Governance and Interoperability Aspects (SMART 2018/0038)." *Shaping Europe's Digital Future - European Commission*,

27 Mar. 2020, ec.europa.eu/digital-single-market/en/news/study-blockchains-legal-governance-and-interoperability-aspects-smart-20180038.

[14] Thartma. "Blockchain Technologies." *Shaping Europe's Digital Future - European Commission*, 15 Jan. 2020, ec.europa.eu/digital-single-market/en/blockchain-technologies.

[15] Ferrari, Valeria. "EU Blockchain Observatory and Forum Workshop on GDPR, Data Policy and Compliance." *SSRN Electronic Journal*, 2018, doi:10.2139/ssrn.3247494.

[16] "EU Blockchain Observatory and Forum and Forum 2018-2020, Conclusions and Reflections.""

[17] Fresno, Blanca González del. "¿Qué Es Un 'Sandbox' Regulatorio?: BBVA." *BBVA NOTICIAS*, BBVA, 20 Nov. 2017, www.bbva.com/es/que-es-un-sandbox-regulatorio/.

[18] "Home." *Ethereum.org*, ethereum.org/en/.

[19] "Hyperledger Fabric." *Hyperledger*, 29 June 2020, www.hyperledger.org/use/fabric.

[20] "La Red." *Alastria*, 6 July 2020, alastria.io/la-red/.

[21] "Hyperledger Besu." *Hyperledger*, 23 Apr. 2020, www.hyperledger.org/use/besu.

[22] "Home." *Enterprise Ethereum Alliance*, 9 Sept. 2020, entethalliance.org/.

[23] Bashir, Imran. *Mastering Blockchain: Distributed Ledger Technology, Decentralization, and Smart Contracts Expained*. Packt, 2018.

[24] "The Pioneer of Proof of Stake." *Peercoin*, www.peercoin.net/.

[25] Parikshit HoodaCheck out this Author's contributed articles., et al. "Practical Byzantine Fault Tolerance(PBFT)." *GeeksforGeeks*, 12 Dec. 2019, www.geeksforgeeks.org/practical-byzantine-fault-tolerancepbft/#:~:text=Practical Byzantine Fault Tolerance is,optimized for low overhead time.

[26] Oza, Mabel. "Practical Byzantine Fault Tolerance (PBFT) the One Consensus to Master." *Medium*, InsatiableMinds, 14 Jan. 2019, medium.com/insatiableminds/practical-byzantine-fault-tolerance-pbft-the-one-consensus-to-master-400397f2b566.

[27] *Hyperledger Chat*, chat.hyperledger.org/channel/fabric-kubernetes.

[28] Laura Esbri, Pau Aragonés. “How to Implement Hyperledger Fabric External Chaincodes within a Kubernetes Cluster.” *Medium*, The Startup, 22 July 2020, medium.com/swlh/how-to-implement-hyperledger-fabric-external-chaincodes-within-a-kubernetes-cluster-fd01d7544523.

[29] “Conceptos.” *Kubernetes*, kubernetes.io/es/docs/concepts/.

[30] “Service.” *Kubernetes*, kubernetes.io/es/docs/concepts/services-networking/service/.

[31] “Volume.” *Kubernetes*, kubernetes.io/es/docs/concepts/storage/volumes/.

[32] “Namespaces.” *Kubernete*s, kubernetes.io/es/docs/concepts/workloads/controllers/deployment/.

[33] “Job.” Kubernetes, kubernetes.io/es/docs/concepts/workloads/controllers/job/.

[34] “Go Is an Open Source Programming Language That Makes It Easy to Build Simple, Reliable, and Efficient Software.” *Go*, golang.org/.

[35] Node.js. *Node.js*, nodejs.org/.

[36] “Seamless Multi-Master Sync, That.” *Apache CouchDB*, couchdb.apache.org/.

[37] “Chaincode for Developers.” *Hyperledger*, hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode4ade.html.

ANEXO

Anexo 1: Coste máquinas virtuales

Anexo 2: API - REST Dockerfile

```
FROM node:8.16-alpine

# Build arguments

ARG NODE_ENV=production

# Create necessary directories.

WORKDIR /app

# Non previlage mode for better security (this user comes with official NodeJS image).

# Copy package.json
```

```
COPY package.json /app

# Copy compiled application files

COPY . /app

# Expose specific port and other necessary ports for debugging

EXPOSE 8080

# Change user to 1022

USER 1022

#Run command

CMD npm run start
```

Anexo 3. Chaincode

```
package main

//import utilities

import(
    "bytes"                                                 //Manipulate byte slices

    "encoding/json"                                          //Encoding and decoding JSON as
RFC7159 (Marshal & Unmarshal functions for golang)

    "fmt"                                                       //Format I/O analogous to C
(printf/scanf)

    "strconv"                                               //Convert to/from string of basic
data types

    //"error"                                                //Manipulate errors

    //"strings"                                              //Manipualte strings

    "time"                                                  //Get actual time to check year

    //API's for the chaincode to access its state variables, transaction context and call
other chaincodes (not calling others cc in this cc)

    //"github.com/hyperledger/fabric/core/chaincode/shim" //OLD PACKAGE

    "github.com/hyperledger/fabric-chaincode-go/shim" //NEW PACKAGE

    //"github.com/hyperledger/fabric/protos/peer" //OLD

    "github.com/hyperledger/fabric-protos-go/peer" //NEW

)


//Definition of the Smart Contract to manage/receive chaincode shim functions

type PassthruChaincode struct {

}


//Definition of the Addendum struct --> json

type AddendumStruct struct {

    CodeId              string `json:"codeId"`                    //Code ID of the addendum
(J-19PS001)

    TemplateId           string `json:"templateId"`                   //Template ID of the
addendum. FE will take into account in order to create the addendum PDF

    Title               string `json:"title"`                     //Title of the addendum

    IdMySQL             string `json:"idMySQL"`                  //Automatic id generated by
the MySQL which has the other attributes of the addendum

    HashDBdata          string `json:"hashDBdata"`               //Hash of the data saved in
the DB. It is saved in the bc to garantee integrity

    Creator              string `json:"creator"`                      //User that starts the
process creating the first record
```

```
    Editor              string `json:"editor"`                   //Editor of this addendum
record
    CommentEditor       string `json:"commentEditor"`            //Comment made by the editor
of this addendum record. Can be null (so no comment)
    Status              string `json:"status"`                   //Status of this addendum
    LastStatus           string `json:"lastStatus"`                //Last Status of this
addendum. By definition will start having N/A (not appliable)
    Bill                string `json:"bill"`                     //Total cost amount of the
addendum
    Currency            string `json:"currency"`                 //Currency type of the bill
    ExpirationDate       string `json:"expirationDate"`            //Expiration date of the
provider offer
    Timestamp            string `json:"timestamp"`                //The time the data is
written
    OffersPDFinfo       []PDFDocStruct `json:"offersPDFinfo"`     //PDFs documents Array of
the addendum.
    ImageInfo            ImageStruct `json:"imageInfo"`            //Image attached to the
addendum doc.
}


//Definition of the addendum update status struct --> json
type AddendumUpdateStatusStruct struct {
    CodeId              string `json:"codeId"`                   //Code ID of the addendum
    Editor              string `json:"editor"`                   //Editor of this addendum
record. Who ordered the transition of the addendum (next status)
    CommentEditor       string `json:"commentEditor"`            //Comment made by the editor
of this addendum record. This can't be null 'cos its a transition'
    Timestamp            string `json:"timestamp"`                //The time the data is
written
}
//Definition of the query struct --> json
type QueryStruct struct {
    Field               string `json:"field"`              //field of the addendum to query
    Value               string `json:"value"`              //value of the field
    SortType            string `json:"sortType"`           //Sort type: asc or desc
}
//Definition of the PDF doc struct --> json
type PDFDocStruct struct {
    PdfSOfilename       string `json:"pdfSOfilename"`       //PDF Filename
    IdNFSpdf            string `json:"idNFSpdf"`           //Path (rute+pdf_hash) where pdf
is saved in the NFS
    HashPDFdata         string `json:"hashPDFdata"`         //Hash of the pdf data storaged
in the NFS. It is saved in the bc to garantee integrity
}
//Definition of the Image struct --> json
type ImageStruct struct {
    ImageFilename       string `json:"imageFilename"`     //Image filename
    ImagePath          string `json:"imagePath"`        //Path (rute+image_hash) where image
is saved in the NFS
    ImageHash           string `json:"imageHash"`         //Hash of the image data storaged
in the NFS. It is saved in the bc to garantee integrity
}
```

```
//Ledger id key name
const LedgerKey = "ledgerkey"


//Ledgerkey json id object: Latest codeId the blockchain has
type LedgerIden struct {
    Id string `json:"id"`
}


//Addendums status
const (
    draft = "DRAFT"
    onReview ="ON_REVIEW"
    approved = "APPROVED"
    completed = "COMPLETED"
    cancelled = "CANCELLED"
    notApplicable = "N/A"
)
//Init of the chaincode
//This function is CALLED when the chaincode is INSTANTIATED or UPGRADED ONCE
//Preparation of the ledger to handle requests and initiate our ledger id
func (t *PassthruChaincode) Init (stub shim.ChaincodeStubInterface) peer.Response {
    fmt.Println("********************** PASSTHRU CHAINCODE INIT **********************")
    //if we want to initialize the ledger with any value or check arguments...
    //In our case we initialize the key value to YYPS000 where YY is the current year (eg 2019 YY=19)
    idJSON, errGet := stub.GetState("ledgerkey") //If the key does not exist in the state database, (nil, nil) is returned.
    if errGet != nil {
        return shim.Error("Init function: ledgerkey - GetState failed")
    } else if len(idJSON) != 0 {    //Checking just one key is enough to know if the chaincode is being initialized or upgraded
        return shim.Success(nil)
    } else {    //Chaincode initialized, we prepare the ledger initializing ledgerkey value to YYPS000
        currentyearArray := splitSubN(strconv.Itoa(time.Now().Year()), 2)  //We get the actual year. E.g. 2019 currentyearArray[1]=19   //19
        //We join year + "PS000" to create the key string
        iden := LedgerIden{(currentyearArray[1] + "PS000")} //000 will not have any addendum
        //Converting a struct into json string byte array
        bytesToLedger, err := json.Marshal(&iden)
        if err != nil {
            return shim.Error("Init function: Could not Marshal ledger init identifier")
        }
        //We store the key and the id in the ledger
        err = stub.PutState(LedgerKey, bytesToLedger)
        if err != nil {
            return shim.Error(fmt.Sprintf("Init function: PutState: Failed to create LedgerKey in the ledger"))
```

```
        }
    }
    //Returns a successful message
    return shim.Success(nil)
}

//Invoke of the Chaincode
//This functions is called when the application requests to run the chaincode.
//arg: arguments needed by the specific function to run.
func (t *PassthruChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    fmt.Println("********************** PASSTHRU CHAINCODE INVOKE **********************")
    //We get the function and arguments from the request
    function, args := stub.GetFunctionAndParameters()
    switch function {
    case "addAddendum":
        return addAddendum(stub, args)
    case "updateAddendum":
        return updateAddendum(stub, args)
    case "getAllAddendums":
        return getAllAddendums(stub)
    case "getAddendum":
        return getAddendum(stub, args)
    case "getOldAddendum":
        return getOldAddendum(stub, args)
    case "getAddendumTemplateId":
        return getAddendumTemplateId(stub, args)
    case "getAddendumHistory":
        return getAddendumHistory(stub, args)
    case "nextAddendumStatus":
        return nextAddendumStatus(stub, args)
    case "returnAddendumToDraft":
        return returnAddendumToDraft(stub, args)
    case "cancelAddendum":
        return cancelAddendum(stub, args)
    case "getAllAddendumsByCreator":
        return getAllAddendumsByCreator(stub, args)
    case "getAllAddendumsByStatus":
        return getAllAddendumsByStatus(stub, args)
    case "getAllAddendumsByCurrency":
        return getAllAddendumsByCurrency(stub, args)
    case "getAllAddendumsExpirationIn7Days":
        return getAllAddendumsExpirationIn7Days(stub)
    case "getAllAddendumsExpirationByDays":
        return getAllAddendumsExpirationByDays(stub, args)
    case "testrichquery": //This will be deleted soon
        return testrichquery(stub, args)
```

```
    }
    //Return error if no switch case or invalid function name
    return shim.Error("Invoke function: Invalid invoke function name: " + function)
}



//Adds a new addendum to the ledger. It will generate what codeId it has, will set the status, lastStatus and Creator and save it along all the addendum params that the input got.
//INPUT: JSON with the following attributes: {IdMySQL, IdNFSpdf, editor, commentEditor, bill, currency, expirationDate and timestamp}
//and the next status of the addendum which can be DRAFT or ON_REVIEW
//Returns a success message or an error
func addAddendum(stub shim.ChaincodeStubInterface, args []string) peer.Response {
    fmt.Println("**********************        PASSTHRU        CHAINCODE        ADDADDENDUM **********************")
    //args     length     =     1     -->     [{"templateId":"01","title":"Esto     es     un titulo","IdMySQL":"123","IdNFSpdf":"456","editor":"me","commentEditor":"you","bill":"123","currency":"123","expirationDate":"123","timestamp":"4-5-19"}]
    //check number of arguments
    if len(args) != 2 {  //arg[0] --> json with parameters args[1] --> next status (we take into account if we save or we save and send to finances)
        fmt.Println(fmt.Sprintf("Invalid  number  of  arguments.  Expected  2  but  got  %d", len(args)))
        return shim.Error(fmt.Sprintf("Invalid number of arguments. Expected 2 but got %d", len(args)))
    }
    //We check the next status given. Can be draft or onReview
    if args[1] != draft && args[1] != onReview {
        fmt.Println(fmt.Sprintf("Creation forbidden because next status is %s and can only be DRAFT or ON_REVIEW", args[1]))
        return shim.Error(fmt.Sprintf("Creation forbidden because next status is %s and can only be DRAFT or ON_REVIEW", args[1]))
    }
    var iden LedgerIden
    //We get the value of the LedgerKey from the ledger. If the ledgerkey does not exist, (nil, nil) is returned
    bytesFromLedger, err := stub.GetState(LedgerKey)
    if err != nil {
        return shim.Error(fmt.Sprintf("Error getting the last state of the LedgerKey with an error of: %s", err))
    }
    if bytesFromLedger == nil {
        return shim.Error("LedgerKey does not exists in the ledger")
    }
    //Unmarshal a JSON byte array into a struct. We have the id in iden.Id
    err = json.Unmarshal(bytesFromLedger, &iden)
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to Unmarshal the JSON byte array - Error %s", err))
    }
    newCodeId := generateCodeId(iden.Id)
    //Error treatment generateCodeId call. By definition all codeIds will have length = 7
```

```
    if len(newCodeId) != 7 { //newCodeId has the error msg then
        fmt.Println(newCodeId)
        return shim.Error(newCodeId)
    }
    //Check addendum arguments
    addendumdata := AddendumStruct{}
    json.Unmarshal([]byte(args[0]), &addendumdata)
    //By definition, we add the new id generated by the blockchain, Status will be args[1],
the first lastStatus will always be notApplicable and Creator will be the first Editor
    addendumdata.CodeId = newCodeId
    addendumdata.Status = args[1]
    addendumdata.LastStatus = notApplicable
    addendumdata.Creator = addendumdata.Editor
    //Checking if the fields of the addendumstruct have correct values
    checks := checkAddendumStruct(addendumdata)
    if checks != "" {
        fmt.Println(checks)
        return shim.Error(checks)
    }
    //Addendum json as bytes to save in the ledger
    addendumAsBytes, errM := json.Marshal(addendumdata)
    if errM != nil {
        fmt.Println(fmt.Sprintf("Error called during json.Marshal with error %s", errM))
        return shim.Error(fmt.Sprintf("Error called during json.Marshal with error %s", errM))
    }
    //Add the new addendum to the ledger
    err = stub.PutState(newCodeId, addendumAsBytes)
    if err != nil {
        fmt.Println(fmt.Sprintf("Error adding an Addendum with codeId " + newCodeId + " to
the ledger with error: %s", err))
        return shim.Error(fmt.Sprintf("Error adding an Addendum with codeId " + newCodeId +
" to the ledger with error: %s", err))
    }
    //We store the generated codeId to the Ledgerkey
    //Marshal a struct into a JSON string byte array
    iden.Id = newCodeId
    bytesToLedger, err := json.Marshal(&iden)
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to Marshal the JSON byte array LedgerKey - Error
%s", err))
    }
    //We store the new key value on the ledger
    err = stub.PutState(LedgerKey, bytesToLedger)
    if err != nil {
        return shim.Error(fmt.Sprintf("Failed to save the value of the key: %s to the ledger
with error %s", LedgerKey, err))
    }
```

```
    //Everything correct, we return a succesful message with codeId
    return shim.Success([]byte(newCodeId))
}



//Update Addendum function
//Updates an addendum to the ledger only if the addendum is in DRAFT state
//INPUT: JSON with the fields we will update the addendum and we need the next status in order to know if it will stay in DRAFT or will go to ON_REVIEW
func updateAddendum(stub shim.ChaincodeStubInterface, args []string) peer.Response {
    fmt.Println("**********************       PASSTHRU       CHAINCODE       UPDATEADDENDUM **********************")
    //check number of arguments
    if len(args) != 2 {  //arg[0] --> json with parameters args[1] --> next status (DRAFT or ON_REVIEW) so 2 arguments
        fmt.Println(fmt.Sprintf("AddAddendum function: Invalid number of arguments. Expected 2 but got %d", len(args)))
        return shim.Error(fmt.Sprintf("AddAddendum function: Invalid number of arguments. Expected 2 but got %d", len(args)))
    }
    //We check the next status given. Can be draft or onReview
    if args[1] != draft && args[1] != onReview {
        fmt.Println(fmt.Sprintf("updateAddendum function: Update forbidden because next status is %s and can only be DRAFT or ON_REVIEW", args[1]))
        return shim.Error(fmt.Sprintf("updateAddendum function: Update forbidden because next status is %s and can only be DRAFT or ON_REVIEW", args[1]))
    }
    //We do not know which attributes are gonna be updated so we transform the json received into a map
    addendumAsMap := make(map[string]string)
    json.Unmarshal([]byte(args[0]), &addendumAsMap)


    //Check if we have a record of that addendum
    addendumGetAsBytes, errGet := stub.GetState(addendumAsMap["codeId"]) //If the key does not exist in the state database, (nil, nil) is returned, so len(addendum)=0
    if errGet != nil {
        return shim.Error("updateAddendum function: GetState failed")
    } else if len(addendumGetAsBytes) == 0 {
        fmt.Println(fmt.Sprintf("updateAddendum function: Key " + addendumAsMap["codeId"] + " does not exist"))
        return     shim.Error(fmt.Sprintf("updateAddendum     function:     Key     "     + addendumAsMap["codeId"] + " does not exist"))
    } else { //It exists, we check if the addendum status = draft (the last version saved in the ledger)
        var addendumOld AddendumStruct
        err := json.Unmarshal(addendumGetAsBytes, &addendumOld)
        if addendumOld.Status != draft {
            fmt.Println(fmt.Sprintf("updateAddendum function: Update forbidden because last addendum status saved in the ledger is %s and you can only update while Status=DRAFT", addendumOld.Status))
            return shim.Error(fmt.Sprintf("updateAddendum function: Update forbidden because last addendum status saved in the ledger is %s and you can only update while Status=DRAFT", addendumOld.Status))
```

```
        }
        //We update the addendum we get from the ledger with the new attributes we received which are in the map
        addendumUpdated, upderr := updateAddendumFromMapToAStruct(addendumOld, addendumAsMap)
        if upderr != "" {
            fmt.Println(fmt.Sprintf("updateAddendum function: Error trying to update the addendum: %s", upderr))
            return shim.Error(fmt.Sprintf("updateAddendum function: Error trying to update the addendum: %s", upderr))
        }
        //We update status and laststatus
        addendumUpdated.LastStatus = addendumOld.Status
        addendumUpdated.Status = args[1]
        //We convert the addendum json to bytes to save in the ledger
        addendumAsBytes, errM := json.Marshal(addendumUpdated)
        if errM != nil {
            fmt.Println(fmt.Sprintf("updateAddendum function: Error called during json.Marshal with error %s", errM))
            return shim.Error(fmt.Sprintf("updateAddendum function: Error called during json.Marshal with error %s", errM))
        }
        //We update the addendum
        err = stub.PutState(addendumUpdated.CodeId, addendumAsBytes)
        if err != nil {
            fmt.Println(fmt.Sprintf("updateAddendum function: Error updating the Addendum: " + addendumUpdated.CodeId + " with error: %s", err))
            return shim.Error(fmt.Sprintf("updateAddendum function: Error update the Addendum: " + addendumUpdated.CodeId + " with error: %s", err))
        }
    }
    //Everything correct, we return a succesful message
    return shim.Success(nil)
}



//Get all Addendums function
//Returns all the addendums the blockchain has (last state) as a JSON byte array ordered by codeId
//This function will be removed in the future for an efficient rich query =) (i think)
func getAllAddendums(stub shim.ChaincodeStubInterface) peer.Response {
    fmt.Println("**********************      PASSTHRU      CHAINCODE      GETALLADDENDUMS **********************")

    //addendums is a JSON array containing all the addendums (last state) the blockchain has
    var addendums bytes.Buffer
    addendums.WriteString("[")
    //ITnow was born in 2012, so we do not contemplate any addendum before 2012. This is how we loop by codeId.
    bArrayMemberAlreadyWritten := false
    for i:=0; i <= time.Now().Year()-2012; i++ { //Get actual year time.Now().Year()
```

```
        yearArray := splitSubN(strconv.Itoa(2012+i),2)

        addendumid := yearArray[1] + "PS001" //By definition we know the first one will be
YYPS001 where YY are the last 2 numbers of the year e.g. 2019 --> 19

        for { //While true in go. We break the loop when we do not get anything in getState

            //Check if we have a record of that addendumid

            addendumAsBytes, errGet := stub.GetState(addendumid) //If the key does not exist
in the state database, (nil, nil) is returned, so len(addendumAsBytes)=0

            if errGet != nil {

                fmt.Println(fmt.Sprintf("getAllAddendums function: GetState failed with id "
+ addendumid))

                return shim.Error(fmt.Sprintf("getAllAddendums function: GetState failed with
id " + addendumid))

            } else if len(addendumAsBytes) == 0 {

                break //go to next year

            } else { //It exists, we save it in the addendums array struct

                var addendum AddendumStruct

                //Unmarshal a JSON byte array into a struct.

                err := json.Unmarshal(addendumAsBytes, &addendum)

                if err != nil {

                    return  shim.Error(fmt.Sprintf("getAllAddendums  function:  Failed  to
Unmarshal the JSON byte array - Error %s", err))

                }

                //Checking if the fields of the addendum have the correct format

                checks := checkAddendumStruct(addendum)

                if checks != "" {

                    fmt.Println(fmt.Sprintf("getAllAddendums function: Error called checking
AddendumStruct with error %s", checks))

                    return  shim.Error(fmt.Sprintf("getAllAddendums  function:  Error  called
checking AddendumStruct with error %s", checks))

                }

                // Add comma before all array members but the first one

                if bArrayMemberAlreadyWritten == true {

                    addendums.WriteString(",")

                }


                addendums.WriteString(string(addendumAsBytes))

                bArrayMemberAlreadyWritten = true


                //we ++1 addendumid to get the next one

                idindexarray    :=    splitSubN(addendumid,4)    //We    get    [19PS    002]
idindexarray[1]=002

                int_indexId, _ := strconv.Atoi(idindexarray[1])

                int_indexId += 1

                next_indexId := strconv.Itoa(int_indexId)

                next_indexId = addZerosToLeftString(next_indexId) //we add zeros to the left
in order to have the same nomenclature

                if len(next_indexId) != 3 { //next_indexId is the error msg if len != 3 so we
pass the error (last valid indexId is 999)

                    fmt.Println(fmt.Sprintf("getAllAddendums function --> %s", next_indexId))

                    return   shim.Error(fmt.Sprintf("getAllAddendums   function   -->   %s",
next_indexId))
```

```
                }
                //We finally create the next one
                addendumid = yearArray[1] + "PS" + next_indexId
            }
        }

    }
    addendums.WriteString("]")

    //We return all the addendums written in a JSON byte array
    return shim.Success(addendums.Bytes())
}



//Gets the attributes of the last state of an addendum given its codeId
//INPUT = codeId --> string
//Returns the addendum (last state) saved on the bc
func getAddendum (stub shim.ChaincodeStubInterface, args []string) peer.Response {
    //check number of arguments
    if len(args) != 1 {  //arg[0] --> codeId of the addendum
        fmt.Println(fmt.Sprintf("getAddendum function: Invalid number of arguments. Expected 1 but got %d", len(args)))
        return shim.Error(fmt.Sprintf("getAddendum function: Invalid number of arguments. Expected 1 but got %d", len(args)))
    }
    //We check if we have a record of that addendum
    addendum, errGet := stub.GetState(args[0]) //If the key does not exist in the state database, (nil, nil) is returned, so len(addendum)=0
    var buffer bytes.Buffer  //Buffer where we will have the addendum. Maybe we can avoid having a buffer and implementing it directly to the retun option
    if errGet != nil {
        return shim.Error(fmt.Sprintf("getAddendum function: GetState failed with error: %s", errGet))
    } else if len(addendum) == 0 {
        fmt.Println(fmt.Sprintf("getAddendum function: Key " + args[0] + " does not exist."))
        return shim.Error(fmt.Sprintf("getAddendum function: Key " + args[0] + " does not exist."))
    } else {
        //It exists, we write the addendum to the buffer
        buffer.WriteString(string(addendum))
        return shim.Success(buffer.Bytes())
    }
}



//Gets the attributes of and old addendum by its codeId and timestamp
//INPUT = codeId --> string, timestamp --> String
//Returns the addendum that has codeId = codeId and timestamp = timestamp
func getOldAddendum (stub shim.ChaincodeStubInterface, args []string) peer.Response {
```

```
    //check number of arguments
    if len(args) != 2 {  //arg[0] --> codeId of the addendum, args[1] --> timestamp
        fmt.Println(fmt.Sprintf("Invalid number of arguments. Expected 2 but got %d", len(args)))
        return shim.Error(fmt.Sprintf("Invalid number of arguments. Expected 2 but got %d", len(args)))
    }

    //we create the query with the codeId and the Timestamp
    queryString := fmt.Sprintf("{\"selector\":{\"codeId\":\"%s\", \"timestamp\":\"%s\"}}", args[0], args[1])
    fmt.Println("INFO de getOldAddendum")
    fmt.Println(fmt.Sprint(queryString))

    queryResults, err := executeQueryAndBuildJSONArray(stub, queryString)
    if err != "" { //If error we return a peer response
        return shim.Error(err)
    }
    //We return the old addendum attributes we found
    fmt.Println(fmt.Sprint("INFO del return de getOldAddendum"))
    fmt.Println(fmt.Sprint(queryResults))
    return shim.Success(queryResults)
}


//Gets the templateId of the Addendum given its codeId
//INPUT = codeId --> string
//Returns the templateId
func getAddendumTemplateId (stub shim.ChaincodeStubInterface, args []string) peer.Response {
    //check number of arguments
    if len(args) != 1 {  //arg[0] --> codeId of the addendum
        fmt.Println(fmt.Sprintf("Invalid number of arguments. Expected 1 but got %d", len(args)))
        return shim.Error(fmt.Sprintf("Invalid number of arguments. Expected 1 but got %d", len(args)))
    }
    //We check if we have a record of that addendum
    addendum, errGet := stub.GetState(args[0]) //If the key does not exist in the state database, (nil, nil) is returned, so len(addendum)=0
    var buffer bytes.Buffer  //Buffer where we will have the addendum. Maybe we can avoid having a buffer and implementing it directly to the retun option
    if errGet != nil {
        return shim.Error(fmt.Sprintf("GetState failed with error: %s", errGet))
    } else if len(addendum) == 0 {
        fmt.Println(fmt.Sprintf("CodeId " + args[0] + " does not exist."))
        return shim.Error(fmt.Sprintf("CodeId " + args[0] + " does not exist."))
    } else {
        //It exists we unmarshal in order to get the templateId
        var addendumStruct AddendumStruct
```

```
        err := json.Unmarshal(addendum, &addendumStruct)
        if err != nil {
            fmt.Println(fmt.Sprintf("Error unmarshalling the Addendum: " + addendumStruct.CodeId + " with error: %s", err))
            return shim.Error(fmt.Sprintf("Error unmarshalling the Addendum: " + addendumStruct.CodeId + " with error: %s", err))
        }
        //We write it to the buffer in order to return
        buffer.WriteString(string(addendumStruct.TemplateId))
        return shim.Success(buffer.Bytes())
    }
}


//Get the history of an addendum
//Returns all the addendums with same id (INPUT: codeId of the addendum you wanna know her story), her history
func getAddendumHistory (stub shim.ChaincodeStubInterface, args []string) peer.Response {
    fmt.Println("**********************      PASSTHRU      CHAINCODE      GETADDENDUMHISTORY **********************")
    //check number of arguments
    if len(args) != 1 {  //arg[0] --> codeId of the addendum
        fmt.Println(fmt.Sprintf("getAddendumHistory function: Invalid number of arguments. Expected 1 but got %d", len(args)))
        return shim.Error(fmt.Sprintf("getAddendumHistory function: Invalid number of arguments. Expected 1 but got %d", len(args)))
    }

    //We check if we have a record of that addendum
    addendum, errGet := stub.GetState(args[0]) //If the key does not exist in the state database, (nil, nil) is returned, so len(addendum)=0
    //Maybe we can do something similar with getHistoryForKey so we avoid 1 stub query, but i cannot test it... who knows :)
    /*
    historyIterator, err := stub.GetHistoryForKey(args[0])
    if err != --> error treatment
    else if err == nil && addendum == nil --> Key not found ????
    else --> dance
    */

    var buffer bytes.Buffer  //Buffer will have the json array with the results
    buffer.WriteString("[")  //We ''start'' the json array

    if errGet != nil {
        return shim.Error(fmt.Sprintf("getAddendumHistory function: GetState failed with error: %s", errGet))
    } else if len(addendum) == 0 {
        fmt.Println(fmt.Sprintf("getAddendumHistory function: Key " + args[0] + " does not exist."))
        return shim.Error(fmt.Sprintf("getAddendumHistory function: Key " + args[0] + " does not exist."))
    } else {
```

```
        //It exists, we get its history
        historyIterator, err := stub.GetHistoryForKey(args[0])
        if err != nil {
            fmt.Println(fmt.Sprintf("getAddendumHistory       function:       Error       in
stub.GetHistoryForKey of " + args[0] + " with error: %s", err))
            return    shim.Error(fmt.Sprintf("getAddendumHistory    function:    Error    in
stub.GetHistoryForKey of " + args[0] + " with error: %s", err))
        }
        defer historyIterator.Close() //Iterator will be closed at the end of the function
thanks to defer =)

        bArrayMemberAlreadyWritten := false
        for historyIterator.HasNext() {
            historyResponse, err := historyIterator.Next()
            if err != nil {
                fmt.Println(fmt.Sprintf("getAddendumHistory         function:         Failed
historyIterator.Next() with Error: %s", err))
                return    shim.Error(fmt.Sprintf("getAddendumHistory    function:    Failed
historyIterator.Next() with Error: %s", err))
            }
            //We add a comma before array members but the first one in order to get a correct
json array
            if bArrayMemberAlreadyWritten == true {
                buffer.WriteString(",")
            }
            bArrayMemberAlreadyWritten = true

            /* No need to check :)
            err = json.Unmarshal(addendumAsBytes.String(), &addendum)
            if err != nil {
                return shim.Error(fmt.Sprintf("getAllAddendums function: Failed to Unmarshal
the JSON byte array - Error %s", err))
            }
            //Checking if the fields of the addendum have the correct format
            checks := checkAddendumStruct(addendum)
            if checks != "" {
                fmt.Println(fmt.Sprintf("getAddendumHistory function: Error called checking
AddendumStruct with error %s", checks))
                return  shim.Error(fmt.Sprintf("getAddendumHistory  function:  Error  called
checking AddendumStruct with error %s", checks))
            }
            */

            //We write the response value
            buffer.WriteString(string(historyResponse.Value))
        }
    }
    buffer.WriteString("]") //We "end" the json array
    //We return all the history of the addendum written in a JSON byte array or [] if noone
    return shim.Success(buffer.Bytes())
```

```
}


//Transitions the addendum to the next state according to the diagram state.
//DRAFT --> ON_REVIEW --> APPROVED --> COMPLETED
//INPUT: arg[0] --> AddendumUpdateStatusStruct {CodeId, Editor, CommentEditor timestamp}
//Returns a succesful msg or an error
func nextAddendumStatus (stub shim.ChaincodeStubInterface, args []string) peer.Response {
    fmt.Println("**********************      PASSTHRU      CHAINCODE      NEXTADDENDUMSTATUS
**********************")
    //check number of arguments
    if len(args) != 1 {    //arg[0] --> AddendumUpdateStatusStruct {CodeId, Editor,
CommentEditor, Timestamp}
        fmt.Println(fmt.Sprintf("nextAddendumStatus function: Invalid number of arguments.
Expected 1 but got %d", len(args)))
        return shim.Error(fmt.Sprintf("nextAddendumStatus function: Invalid number of
arguments. Expected 1 but got %d", len(args)))
    }
    //Unmarshal the addendumUpdateStatusS in order to receive the attributes we need to update
    addendumUpdateStatusS:= AddendumUpdateStatusStruct{}
    json.Unmarshal([]byte(args[0]), &addendumUpdateStatusS)
    checks := checkAddendumUpdateStatusStruct(addendumUpdateStatusS)
    if checks != "" {
        fmt.Println(fmt.Sprintf("nextAddendumStatus function: Error called checking
checkAddendumUpdateStatusStruct with error %s", checks))
        return shim.Error(fmt.Sprintf("nextAddendumStatus function: Error called checking
checkAddendumUpdateStatusStruct with error %s", checks))
    }
    //Check if we have a record of that addendum
    addendumFromLedgerAsBytes, errGet := stub.GetState(addendumUpdateStatusS.CodeId) //If the
key does not exist in the state database, (nil, nil) is returned, so len(addendum)=0
    if errGet != nil {
        return shim.Error("nextAddendumStatus function: GetState failed")
    } else if len(addendumFromLedgerAsBytes) == 0 {
        fmt.Println(fmt.Sprintf("nextAddendumStatus      function:      Key      "      +
addendumUpdateStatusS.CodeId + " does not exist."))
        return    shim.Error(fmt.Sprintf("nextAddendumStatus    function:    Key    "    +
addendumUpdateStatusS.CodeId + " does not exist."))
    } else { //It exists, we get the struct in order to update the state
        addendum:= AddendumStruct{}
        json.Unmarshal(addendumFromLedgerAsBytes, &addendum)
        //Checking if the fields of the addendumstruct have correct values
        checks := checkAddendumStruct(addendum)
        if checks != "" {
            fmt.Println(fmt.Sprintf("nextAddendumStatus function: Error called checking
AddendumStruct with error %s", checks))
            return shim.Error(fmt.Sprintf("nextAddendumStatus function: Error called checking
AddendumStruct with error %s", checks))
        }
        //We update the status
        nextStatus, errUS := updateStatus(addendum.Status)
```

```
        if errUS != "" {
            fmt.Println(fmt.Sprintf("nextAddendumStatus function: Error called updating status in %s", errUS))
            return shim.Error(fmt.Sprintf("nextAddendumStatus function: Error called updating status in %s", errUS))
        }
        //We update status, last status, editor and commenteditor
        addendum.LastStatus = addendum.Status
        addendum.Status = nextStatus
        addendum.Editor = addendumUpdateStatusS.Editor
        addendum.CommentEditor = addendumUpdateStatusS.CommentEditor
        addendum.Timestamp = addendumUpdateStatusS.Timestamp

        //Addendum json as bytes to save in the ledger
        addendumAsBytes, errM := json.Marshal(addendum)
        if errM != nil {
            fmt.Println(fmt.Sprintf("nextAddendumStatus function: Error called during json.Marshal with error %s", errM))
            return shim.Error(fmt.Sprintf("nextAddendumStatus function: Error called during json.Marshal with error %s", errM))
        }
        //Update the addendum
        err := stub.PutState(addendumUpdateStatusS.CodeId, addendumAsBytes)
        if err != nil {
            fmt.Println(fmt.Sprintf("nextAddendumStatus function: Error updating the Addendum: " + addendumUpdateStatusS.CodeId + " with error: %s", err))
            return shim.Error(fmt.Sprintf("nextAddendumStatus function: Error update the Addendum: " + addendumUpdateStatusS.CodeId + " with error: %s", err))
        }
    }
    //Everything correct, we return a succesful message
    return shim.Success(nil)
}



//Transitions the addendum to the DRAFT state. Only can go back to draft if its actual state is ON_REVIEW
//INPUT: addendumUpdateStatusS.CodeId {codeId, Editor, CommentEditor, Timestamp}
//Returns a succesful msg or an error
func returnAddendumToDraft (stub shim.ChaincodeStubInterface, args []string) peer.Response {
    fmt.Println("**********************     PASSTHRU     CHAINCODE     RETURNADDENDUMTODRAFT **********************")
    //check number of arguments
    if len(args) != 1 {  //arg[0] --> addendumUpdateStatusS of the next status
        fmt.Println(fmt.Sprintf("returnAddendumToDraft function: Invalid number of arguments. Expected 1 but got %d", len(args)))
        return shim.Error(fmt.Sprintf("returnAddendumToDraft function: Invalid number of arguments. Expected 1 but got %d", len(args)))
    }
    //Unmarshal the addendumUpdateStatusS in order to receive the attributes we need to update
```

```
    addendumUpdateStatusS:= AddendumUpdateStatusStruct{}
    json.Unmarshal([]byte(args[0]), &addendumUpdateStatusS)
    checks := checkAddendumUpdateStatusStruct(addendumUpdateStatusS)
    if checks != "" {
        fmt.Println(fmt.Sprintf("returnAddendumToDraft function: Error called checking checkAddendumUpdateStatusStruct with error %s", checks))
        return shim.Error(fmt.Sprintf("returnAddendumToDraft function: Error called checking checkAddendumUpdateStatusStruct with error %s", checks))
    }
    //Check if we have a record of that addendum
    addendumFromLedgerAsBytes, errGet := stub.GetState(addendumUpdateStatusS.CodeId) //If the key does not exist in the state database, (nil, nil) is returned, so len(addendum)=0
    if errGet != nil {
        return shim.Error("returnAddendumToDraft function: GetState failed")
    } else if len(addendumFromLedgerAsBytes) == 0 {
        fmt.Println(fmt.Sprintf("returnAddendumToDraft function: Key " + addendumUpdateStatusS.CodeId + " does not exist."))
        return shim.Error(fmt.Sprintf("returnAddendumToDraft function: Key " + addendumUpdateStatusS.CodeId + " does not exist."))
    } else { //It exists, we get the struct in order to update the state
        addendum:= AddendumStruct{}
        json.Unmarshal(addendumFromLedgerAsBytes, &addendum)
        //Checking if the fields of the addendumstruct have correct values
        checks := checkAddendumStruct(addendum)
        if checks != "" {
            fmt.Println(fmt.Sprintf("returnAddendumToDraft function: Error called checking AddendumStruct with error %s", checks))
            return shim.Error(fmt.Sprintf("returnAddendumToDraft function: Error called checking AddendumStruct with error %s", checks))
        }
        //We check if the addendum is onReview
        if addendum.Status != onReview {
            fmt.Println(fmt.Sprintf("returnAddendumToDraft function: Cannot return to draft status. Actual status is %s", addendum.Status))
            return shim.Error(fmt.Sprintf("returnAddendumToDraft function: Cannot return to draft status. Actual status is %s", addendum.Status))
        }
        //We update status, last status, editor and comment editor of the addendum
        addendum.LastStatus = addendum.Status
        addendum.Status = draft
        addendum.Editor = addendumUpdateStatusS.Editor
        addendum.CommentEditor = addendumUpdateStatusS.CommentEditor
        addendum.Timestamp = addendumUpdateStatusS.Timestamp
        //Addendum json as bytes to save in the ledger
        addendumAsBytes, errM := json.Marshal(addendum)
        if errM != nil {
            fmt.Println(fmt.Sprintf("returnAddendumToDraft function: Error called during json.Marshal with error %s", errM))
            return shim.Error(fmt.Sprintf("returnAddendumToDraft function: Error called during json.Marshal with error %s", errM))
        }
```

```
        //Update the addendum
        err := stub.PutState(addendumUpdateStatusS.CodeId, addendumAsBytes)
        if err != nil {
            fmt.Println(fmt.Sprintf("returnAddendumToDraft function: Error updating the Addendum: " + addendumUpdateStatusS.CodeId + " with error: %s", err))
            return shim.Error(fmt.Sprintf("returnAddendumToDraft function: Error update the Addendum: " + addendumUpdateStatusS.CodeId + " with error: %s", err))
        }
    }
    //Everything correct, we return a succesful message
    return shim.Success(nil)
}



//Transitions the addendum to the CANCELLED state. Only can go to CANCELLED state if its actual state
//is DRAFT, ON_REVIEW or APPROVED.
//INPUT: addendumUpdateStatusS {codeId, Editor, CommentEditor, Timestamp}
//Returns a succesful msg or an error
func cancelAddendum (stub shim.ChaincodeStubInterface, args []string) peer.Response {
    fmt.Println("**********************       PASSTHRU       CHAINCODE       CANCELADDENDUM **********************")
    //check number of arguments
    if len(args) != 1 {  //arg[0] --> addendumUpdateStatusS of the update
        fmt.Println(fmt.Sprintf("cancelAddendum function: Invalid number of arguments. Expected 1 but got %d", len(args)))
        return shim.Error(fmt.Sprintf("cancelAddendum function: Invalid number of arguments. Expected 1 but got %d", len(args)))
    }
    //Unmarshal the addendumUpdateStatusS in order to receive the attributes we need to update
    addendumUpdateStatusS:= AddendumUpdateStatusStruct{}
    json.Unmarshal([]byte(args[0]), &addendumUpdateStatusS)
    checks := checkAddendumUpdateStatusStruct(addendumUpdateStatusS)
    if checks != "" {
        fmt.Println(fmt.Sprintf("cancelAddendum function: Error called checking checkAddendumUpdateStatusStruct with error %s", checks))
        return shim.Error(fmt.Sprintf("cancelAddendum function: Error called checking checkAddendumUpdateStatusStruct with error %s", checks))
    }
    //Check if we have a record of that addendum
    addendumFromLedgerAsBytes, errGet := stub.GetState(addendumUpdateStatusS.CodeId) //If the key does not exist in the state database, (nil, nil) is returned, so len(addendum)=0
    if errGet != nil {
        return shim.Error("cancelAddendum function: GetState failed")
    } else if len(addendumFromLedgerAsBytes) == 0 {
        fmt.Println(fmt.Sprintf("cancelAddendum function: Key " + addendumUpdateStatusS.CodeId + " does not exist."))
        return shim.Error(fmt.Sprintf("cancelAddendum function: Key " + addendumUpdateStatusS.CodeId + " does not exist."))
    } else { //It exists, we get the struct in order to update the state
        addendum:= AddendumStruct{}
```

```
        json.Unmarshal(addendumFromLedgerAsBytes, &addendum)
        //Checking if the fields of the addendumstruct have correct values
        checks := checkAddendumStruct(addendum)
        if checks != "" {
            fmt.Println(fmt.Sprintf("cancelAddendum function: Error called checking AddendumStruct with error %s", checks))
            return shim.Error(fmt.Sprintf("cancelAddendum function: Error called checking AddendumStruct with error %s", checks))
        }
        //We cancel the addendum only if is not with status COMPLETED
        if addendum.Status == completed || addendum.Status == cancelled {
            fmt.Println(fmt.Sprintf("cancelAddendum function: Cannot cancel the addendum. Actual status is %s", addendum.Status))
            return shim.Error(fmt.Sprintf("cancelAddendum function: Cannot cancel the addendum. Actual status is %s", addendum.Status))
        }
        //We update the status, last status, editor and comment editor
        addendum.LastStatus = addendum.Status
        addendum.Status = cancelled
        addendum.Editor = addendumUpdateStatusS.Editor
        addendum.CommentEditor = addendumUpdateStatusS.CommentEditor
        addendum.Timestamp = addendumUpdateStatusS.Timestamp

        //Addendum json as bytes to save in the ledger
        addendumAsBytes, errM := json.Marshal(addendum)
        if errM != nil {
            fmt.Println(fmt.Sprintf("cancelAddendum function: Error called during json.Marshal with error %s", errM))
            return shim.Error(fmt.Sprintf("cancelAddendum function: Error called during json.Marshal with error %s", errM))
        }
        //Update the addendum
        err := stub.PutState(addendumUpdateStatusS.CodeId, addendumAsBytes)
        if err != nil {
            fmt.Println(fmt.Sprintf("cancelAddendum function: Error updating the Addendum: " + addendumUpdateStatusS.CodeId + " with error: %s", err))
            return shim.Error(fmt.Sprintf("cancelAddendum function: Error update the Addendum: " + addendumUpdateStatusS.CodeId + " with error: %s", err))
        }
    }
    //Everything correct, we return a succesful message
    return shim.Success(nil)
}



//Executes the query (we got from INPUT) to the blockchain.
//INPUT = a rich query string (COUCHDB COMPLIANCE) --> e.g. "{\"selector\":{\"god\":\"pol\"}}"
//Returns a JSON array containing all the results the query got
func executeQueryAndBuildJSONArray (stub shim.ChaincodeStubInterface, query string) ([]byte, string) {
```

```
    fmt.Println("**********************  PASSTHRU  CHAINCODE  EXECUTEQUERYANDBUILDJSONARRAY
**********************")


    resultsIterator, err := stub.GetQueryResult(query)  //we get an iterator having all the
query results we got

    if err != nil { //We check i we had an error

        fmt.Println(fmt.Sprintf("executeQueryAndBuildJSONArray  function:  Error  executing
GetQueryResult with error: %s", err))

        return  nil,  fmt.Sprintf("executeQueryAndBuildJSONArray  function:  Error  executing
GetQueryResult with error: %s", err)

    }

    defer resultsIterator.Close() //we close the iteraror. It will be closed at the end of
the function thanks to defer =)


    var buffer bytes.Buffer  //Buffer will have the json array with the results

    buffer.WriteString("[")  //We ''start'' the json array


    bArrayMemberAlreadyWritten := false

    for resultsIterator.HasNext() {

        queryResponse, err := resultsIterator.Next()

        if err != nil {

            fmt.Println(fmt.Sprintf("executeQueryAndBuildJSONArray      function:      Error
resultsIterator.next() failed with error: %s", err))

            return    nil,    fmt.Sprintf("executeQueryAndBuildJSONArray    function:    Error
resultsIterator.next() failed with error: %s", err)

        }

        //We add a comma before array members but the first one in order to get a correct
json array

        if bArrayMemberAlreadyWritten == true {

            buffer.WriteString(",")

        }

        bArrayMemberAlreadyWritten = true

        fmt.Println(fmt.Sprintf(queryResponse.Key))

        fmt.Println(fmt.Sprintf(string(queryResponse.Value)))

        //We write the response value

        buffer.WriteString(string(queryResponse.Value))

    }

    buffer.WriteString("]") //We "end" the json array


    return buffer.Bytes(), ""

}




//Get all the addendums (last state) where creator = INPUT

//INPUT = u2dXXXXX (string)

//Returns a JSON byte array with all the addendums with CREATOR=INPUT or void [] if noone

func getAllAddendumsByCreator (stub shim.ChaincodeStubInterface, args []string) peer.Response
{
```

```
    fmt.Println("**********************    PASSTHRU    CHAINCODE    GETALLADDENDUMBYCREATOR
**********************")
    //check number of arguments
    if len(args) != 1 {  //arg[0] --> creator u2d
        fmt.Println(fmt.Sprintf("getAllAddendumsByCreator   function:   Invalid   number   of
arguments. Expected 1 but got %d", len(args)))
        return shim.Error(fmt.Sprintf("getAllAddendumsByCreator function: Invalid number of
arguments. Expected 1 but got %d", len(args)))
    }
    fmt.Println(fmt.Sprintf("Creator input: %s", args[0]))
    //We check if it is a correct creator (atm only we check if it is not an empty string)
    err := checkCreator(args[0])
    if err != "" {
        return shim.Error(err)
    }
    //"{\"selector\":{\"creator\":\"pol\"}}"
    //we create the query with the creator we got
    queryString := fmt.Sprintf("{\"selector\":{\"creator\":\"%s\"}}", args[0])

    queryResults, err := executeQueryAndBuildJSONArray(stub, queryString)
    if err != "" { //If error we return a peer response
        return shim.Error(err)
    }
    //We return the result JSON array
    return shim.Success(queryResults)
}



//Get all the addendums (last state) where status = INPUT
//INPUT = STATUS (string)
//Returns a JSON byte array with all the addendums with STATUS=INPUT or void [] if noone
func getAllAddendumsByStatus (stub shim.ChaincodeStubInterface, args []string) peer.Response
{
    fmt.Println("**********************    PASSTHRU    CHAINCODE    GETALLADDENDUMBYSTATUS
**********************")
    //check number of arguments
    if len(args) != 1 {  //arg[0] --> status
        fmt.Println(fmt.Sprintf("getAllAddendumsByStatus   function:   Invalid   number   of
arguments. Expected 1 but got %d", len(args)))
        return shim.Error(fmt.Sprintf("getAllAddendumsByStatus function: Invalid number of
arguments. Expected 1 but got %d", len(args)))
    }
    //We check if its a correct status
    err := checkStatus(args[0])
    if err != "" {
        return shim.Error(err)
    }
    //we create the query with the status we got
    queryString := fmt.Sprintf("{\"selector\":{\"status\":\"%s\"}}", args[0])
```

```
    queryResults, err := executeQueryAndBuildJSONArray(stub, queryString)
    if err != "" { //If error we return a peer response
        return shim.Error(err)
    }
    //We return the result JSON array
    return shim.Success(queryResults)
}



//Get all the addendums (last state) where currency = INPUT
//INPUT = STATUS (string --> € o $)
//Returns a JSON byte array with all the addendums with CURRENCY=INPUT or void [] if noone
func getAllAddendumsByCurrency (stub shim.ChaincodeStubInterface, args []string) peer.Response {
    fmt.Println("**********************  PASSTHRU  CHAINCODE  GETALLADDENDUMBYCURRENCY **********************")
    //check number of arguments
    if len(args) != 1 {  //arg[0] --> status
        fmt.Println(fmt.Sprintf("Invalid number of arguments. Expected 1 but got %d", len(args)))
        return shim.Error(fmt.Sprintf("Invalid number of arguments. Expected 1 but got %d", len(args)))
    }
    //We check if it is a correct currency
    err := checkCurrency(args[0])
    if err != "" {
        return shim.Error(err)
    }
    //we create the query with the status we got
    queryString := fmt.Sprintf("{\"selector\":{\"currency\":\"%s\"}}", args[0])

    queryResults, err := executeQueryAndBuildJSONArray(stub, queryString)
    if err != "" { //If error we return a peer response
        return shim.Error(err)
    }
    //We return the result JSON array
    return shim.Success(queryResults)
}



//Get all the addendums (last state) that will expire its suppleir offer in the next 7 days (we consider today as the first day of these 7)
//Returns a JSON byte array with all the addendums that will expire its supplier offer in the next 7 days or void if noone
func getAllAddendumsExpirationIn7Days (stub shim.ChaincodeStubInterface) peer.Response {
    fmt.Println("**********************  PASSTHRU  CHAINCODE GETALLADDENDUMEXPIRATIONDATEIN7DAYS **********************")
    //We create the array having the 7 days of expirationDate we are looking for
```

```
    //This is not a beauty way but works.
    //May think a little bit to improve efficiency in this query =)
    sevendays := create7daysArray()

    //we create the query with the array of days we got
    queryString     :=     fmt.Sprintf("{\"selector\":{\"expirationDate\":{\"$in\":[\""     +
sevendays[0] + "\", \"" + sevendays[1] + "\", \"" + sevendays[2] + "\", \"" + sevendays[3] +
"\", \"" + sevendays[4] + "\", \"" + sevendays[5] + "\", \"" + sevendays[6] + "\"]}}}")

    queryResults, err := executeQueryAndBuildJSONArray(stub, queryString)
    if err != "" { //If error we return a peer response
        return shim.Error(err)
    }
    //We return the result JSON array
    return shim.Success(queryResults)
}

//Get all the addendums (last state) that will expire its suppleir offer in the next 7 days
(we consider today as the first day of these 7)
//Returns a JSON byte array with all the addendums that will expire its supplier offer in the
next 7 days or void if noone
func  getAllAddendumsExpirationByDays  (stub  shim.ChaincodeStubInterface,  args  []string)
peer.Response {
    fmt.Println("**********************  PASSTHRU  CHAINCODE  GETALLADDENDUMEXPIRATIONBYDAYS
**********************")
    //check number of arguments
    if len(args) != 1 {  //args[0] --> number of days
        fmt.Println(fmt.Sprintf("Invalid  number  of  arguments.  Expected  1  but  got  %d",
len(args)))
        return shim.Error(fmt.Sprintf("Invalid number of arguments. Expected 1 but got %d",
len(args)))
    }
    //We create the array given the input number of days
    days_int, _ := strconv.Atoi(args[0])
    if days_int < 2 {
        fmt.Println(fmt.Sprintf("Parameter  Days  of  getAllAddendumsExpirationByDays  must  be
greater than 1"))
        return shim.Error(fmt.Sprintf("Parameter Days of getAllAddendumsExpirationByDays must
be greater than 1"))
    }
    days := createDaysArray(days_int)

    //we create the query with the array of days we got, add the first day and we start the
loop if need it
    queryString := fmt.Sprintf("{\"selector\":{\"expirationDate\":{\"$in\":[\"" + days[0])
    for i:=1; i<len(days); i++ {
        //[\"" + sevendays[0] + "\", \"" + sevendays[1] + "\", \""
        //sevendays[5] + "\", \"" + sevendays[6] + "\"]}}}")
        queryString = fmt.Sprintf(queryString + "\", \"" + days[i])
    }
    //We close the querySring
```

```
    queryString = fmt.Sprintf(queryString + "\"]}}}")

    queryResults, err := executeQueryAndBuildJSONArray(stub, queryString)
    if err != "" { //If error we return a peer response
        return shim.Error(err)
    }
    //We return the result JSON array
    return shim.Success(queryResults)
}
//Function to test rich queries from cli
//Will be deleted =)
func testrichquery (stub shim.ChaincodeStubInterface, args []string) peer.Response {
    queryResults, err := executeQueryAndBuildJSONArray(stub, args[0])
    if err != "" { //If error we return a peer response
        return shim.Error(err)
    }
    //We return the result JSON array
    return shim.Success(queryResults)
}
//Main function
//Starts up the chaincode in the container during instantiate
func main() {
    fmt.Println("********************** PASSTHRU CHAINCODE MAIN **********************")
    //The ccid is assigned to the chaincode on install (using the "peer lifecycle chaincode
install <package>" command) for instance

    server := &shim.ChaincodeServer{
        CCID:    os.Getenv("CHAINCODE_CCID"),
        Address: os.Getenv("CHAINCODE_ADDRESS"),
        CC:      new(PassthruChaincode),
        TLSProps: shim.TLSProperties{
                Disabled: true,
        },
    }

    // Start the chaincode external server
    err := server.Start()

    if err != nil {
        fmt.Printf("Main function: Error creating the PassthruChaincode: %s", err)
    }
}
```

Anexo 4. API -REST Dockerfile productive

```
FROM /…/registry.redhat.io/rhel8/nodejs-12:1-27
#FROM registry.redhat.io/rhel8/nodejs-12
```

```
WORKDIR /opt/app-root/src

# Bundle app source

COPY src/ .

# descargamos los modulos de node del nexus

#ARG       CHAIN_REPO="https://nexuscorp.lacaixa.es:8443/nexus/repository/cbk-cloud-generic-artifacts/PCLD/PASBCK"

#ARG CHAIN_EXE="modules.tar.gz"

#RUN wget "${CHAIN_REPO}"/"${CHAIN_EXE}" --no-check-certificate

COPY ./modules.tar.gz .

RUN mkdir node_modules

RUN tar -xvf modules.tar.gz node_modules/

RUN rm -f modules.tar.gz

# RUN npm install # no podemos hacer esto porque no hay salida a internet

# If you are building your code for production

# RUN npm ci --only=production

#RUN mkdir api-data

EXPOSE 8443

USER 1022

CMD [ "npm", "start"]
```

Anexo 5. Hyperledgr Fabric ficheros de configuración de Kubernetes

CouchDB

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: couchdb-org0alfa
  name: couchdb-org0alfa
  namespace: new-bck
spec:
  ports:
  - name: couchdb-org0alfa
    port: 5984
    targetPort: 5984
  selector:
    app: couchdb-org0alfa
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: couchdb-org0alfa
```

`

```
  name: couchdb-org0alfa
  namespace: new-bck
spec:
  selector:
    matchLabels:
      app: couchdb-org0alfa
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: couchdb-org0alfa
        logging.elk.stack: platform
    spec:
      containers:
      - env:
        - name: COUCHDB_USER
          value: user0
        - name: COUCHDB_PASSWORD
          value: pass0
        name: couchdb-org0alfa
        image: fabric-couch:2.0.1
        resources:
          requests:
            memory: "64Mi"
            cpu: "50m"
          limits:
            memory: "500Mi"
            cpu: "256m"
        ports:
          - containerPort: 5984
        volumeMounts:
        - name: couchdb-org0alfa-persistentdata
          mountPath: /opt/couchdb/data
      volumes:
      - name: couchdb-org0alfa-persistentdata
        persistentVolumeClaim:
          claimName: pasbck-couchdb0alfa-new
---
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pasbck-couchdb0alfa-new
  namespace: new-bck
  annotations:
```

```
    pv.beta.kubernetes.io/gid: "0"
  labels:
    name: pasbck-couchdb0alfa-new
spec:
  capacity:
    storage: 250Mi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/mnt/new-bck/hyperledger-network/fabric-persistent-data/couchdb0alfa"
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pasbck-couchdb0alfa-new
  namespace: new-bck
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 250Mi
  selector:
    matchLabels:
      name: pasbck-couchdb0alfa-new
---
```

Peer0A

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: builders-config
  namespace: new-bck
  labels:
    app: builders-config
data:
  core.yaml: |
    # Copyright IBM Corp. All Rights Reserved.
    #
    # SPDX-License-Identifier: Apache-2.0
    #


############################################################################
    #
    #    Peer section
    #

############################################################################
    peer:

        # The peer id provides a name for this peer instance and is used when
 `
```

```
        # naming docker resources.
        id: jdoe

        # The networkId allows for logical separation of networks and is used
when
        # naming docker resources.
        networkId: dev

        # The Address at local network interface this Peer will listen on.
        # By default, it will listen on all network interfaces
        listenAddress: 0.0.0.0:7051

        # The endpoint this peer uses to listen for inbound chaincode
connections.
        # If this is commented-out, the listen address is selected to be
        # the peer's address (see below) with port 7052
        # chaincodeListenAddress: 0.0.0.0:7052

        # The endpoint the chaincode for this peer uses to connect to the peer.
        # If this is not specified, the chaincodeListenAddress address is
selected.
        # And if chaincodeListenAddress is not specified, address is selected
from
        # peer listenAddress.
        # chaincodeAddress: 0.0.0.0:7052

        # When used as peer config, this represents the endpoint to other peers
        # in the same organization. For peers in other organization, see
        # gossip.externalEndpoint for more info.
        # When used as CLI config, this means the peer's endpoint to interact
with
        address: 0.0.0.0:7051

        # Whether the Peer should programmatically determine its address
        # This case is useful for docker containers.
        addressAutoDetect: false

        # Keepalive settings for peer server and clients
        keepalive:
            # Interval is the duration after which if the server does not see
            # any activity from the client it pings the client to see if it's
alive
            interval: 7200s
            # Timeout is the duration the server waits for a response
            # from the client after sending a ping before closing the connection
            timeout: 20s
            # MinInterval is the minimum permitted time between client pings.
            # If clients send pings more frequently, the peer server will
            # disconnect them
            minInterval: 60s
            # Client keepalive settings for communicating with other peer nodes
            client:
                # Interval is the time between pings to peer nodes.  This must
                # greater than or equal to the minInterval specified by peer
                # nodes
                interval: 60s
                # Timeout is the duration the client waits for a response from
                # peer nodes before closing the connection
                timeout: 20s
```

```
            # DeliveryClient keepalive settings for communication with ordering
            # nodes.
            deliveryClient:
                # Interval is the time between pings to ordering nodes.  This
must
                # greater than or equal to the minInterval specified by ordering
                # nodes.
                interval: 60s
                # Timeout is the duration the client waits for a response from
                # ordering nodes before closing the connection
                timeout: 20s


        # Gossip related configuration
        gossip:
            # Bootstrap set to initialize gossip with.
            # This is a list of other peers that this peer reaches out to at
startup.
            # Important: The endpoints here have to be endpoints of peers in the
same
            # organization, because the peer would refuse connecting to these
endpoints
            # unless they are in the same organization as the peer.
            bootstrap: 127.0.0.1:7051

            # NOTE: orgLeader and useLeaderElection parameters are mutual
exclusive.
            # Setting both to true would result in the termination of the peer
            # since this is undefined state. If the peers are configured with
            # useLeaderElection=false, make sure there is at least 1 peer in the
            # organization that its orgLeader is set to true.

            # Defines whenever peer will initialize dynamic algorithm for
            # "leader" selection, where leader is the peer to establish
            # connection with ordering service and use delivery protocol
            # to pull ledger blocks from ordering service. It is recommended to
            # use leader election for large networks of peers.
            useLeaderElection: true
            # Statically defines peer to be an organization "leader",
            # where this means that current peer will maintain connection
            # with ordering service and disseminate block across peers in
            # its own organization
            orgLeader: false

            # Interval for membershipTracker polling
            membershipTrackerInterval: 5s

            # Overrides the endpoint that the peer publishes to peers
            # in its organization. For peers in foreign organizations
            # see 'externalEndpoint'
            endpoint:
            # Maximum count of blocks stored in memory
            maxBlockCountToStore: 100
            # Max time between consecutive message pushes(unit: millisecond)
            maxPropagationBurstLatency: 10ms
            # Max number of messages stored until a push is triggered to remote
peers
            maxPropagationBurstSize: 10
            # Number of times a message is pushed to remote peers
```

```
            propagateIterations: 1
            # Number of peers selected to push messages to
            propagatePeerNum: 3
            # Determines frequency of pull phases(unit: second)
            # Must be greater than digestWaitTime + responseWaitTime
            pullInterval: 4s
            # Number of peers to pull from
            pullPeerNum: 3
            # Determines frequency of pulling state info messages from
peers(unit: second)
            requestStateInfoInterval: 4s
            # Determines frequency of pushing state info messages to peers(unit:
second)
            publishStateInfoInterval: 4s
            # Maximum time a stateInfo message is kept until expired
            stateInfoRetentionInterval:
            # Time from startup certificates are included in Alive messages(unit:
second)
            publishCertPeriod: 10s
            # Should we skip verifying block messages or not (currently not in
use)
            skipBlockVerification: false
            # Dial timeout(unit: second)
            dialTimeout: 3s
            # Connection timeout(unit: second)
            connTimeout: 2s
            # Buffer size of received messages
            recvBuffSize: 20
            # Buffer size of sending messages
            sendBuffSize: 200
            # Time to wait before pull engine processes incoming digests (unit:
second)
            # Should be slightly smaller than requestWaitTime
            digestWaitTime: 1s
            # Time to wait before pull engine removes incoming nonce (unit:
milliseconds)
            # Should be slightly bigger than digestWaitTime
            requestWaitTime: 1500ms
            # Time to wait before pull engine ends pull (unit: second)
            responseWaitTime: 2s
            # Alive check interval(unit: second)
            aliveTimeInterval: 5s
            # Alive expiration timeout(unit: second)
            aliveExpirationTimeout: 25s
            # Reconnect interval(unit: second)
            reconnectInterval: 25s
            # This is an endpoint that is published to peers outside of the
organization.
            # If this isn't set, the peer will not be known to other
organizations.
            externalEndpoint:
            # Leader election service configuration
            election:
                # Longest time peer waits for stable membership during leader
election startup (unit: second)
                startupGracePeriod: 15s
                # Interval gossip membership samples to check its stability
(unit: second)
                membershipSampleInterval: 1s
```

```
                # Time passes since last declaration message before peer decides
to perform leader election (unit: second)
                leaderAliveThreshold: 10s
                # Time between peer sends propose message and declares itself as
a leader (sends declaration message) (unit: second)
                leaderElectionDuration: 5s

            pvtData:
                # pullRetryThreshold determines the maximum duration of time
private data corresponding for a given block
                # would be attempted to be pulled from peers until the block
would be committed without the private data
                pullRetryThreshold: 60s
                # As private data enters the transient store, it is associated
with the peer's ledger's height at that time.
                # transientstoreMaxBlockRetention defines the maximum difference
between the current ledger's height upon commit,
                # and the private data residing inside the transient store that
is guaranteed not to be purged.
                # Private data is purged from the transient store when blocks
with sequences that are multiples
                # of transientstoreMaxBlockRetention are committed.
                transientstoreMaxBlockRetention: 1000
                # pushAckTimeout is the maximum time to wait for an
acknowledgement from each peer
                # at private data push at endorsement time.
                pushAckTimeout: 3s
                # Block to live pulling margin, used as a buffer
                # to prevent peer from trying to pull private data
                # from peers that is soon to be purged in next N blocks.
                # This helps a newly joined peer catch up to current
                # blockchain height quicker.
                btlPullMargin: 10
                # the process of reconciliation is done in an endless loop, while
in each iteration reconciler tries to
                # pull from the other peers the most recent missing blocks with a
maximum batch size limitation.
                # reconcileBatchSize determines the maximum batch size of missing
private data that will be reconciled in a
                # single iteration.
                reconcileBatchSize: 10
                # reconcileSleepInterval determines the time reconciler sleeps
from end of an iteration until the beginning
                # of the next reconciliation iteration.
                reconcileSleepInterval: 1m
                # reconciliationEnabled is a flag that indicates whether private
data reconciliation is enable or not.
                reconciliationEnabled: true
                # skipPullingInvalidTransactionsDuringCommit is a flag that
indicates whether pulling of invalid
                # transaction's private data from other peers need to be skipped
during the commit time and pulled
                # only through reconciler.
                skipPullingInvalidTransactionsDuringCommit: false

            # Gossip state transfer related configuration
            state:
                # indicates whenever state transfer is enabled or not
                # default value is true, i.e. state transfer is active
```

```
                # and takes care to sync up missing blocks allowing
                # lagging peer to catch up to speed with rest network
                enabled: true
                # checkInterval interval to check whether peer is lagging behind
enough to
                # request blocks via state transfer from another peer.
                checkInterval: 10s
                # responseTimeout amount of time to wait for state transfer
response from
                # other peers
                responseTimeout: 3s
                # batchSize the number of blocks to request via state transfer
from another peer
                batchSize: 10
                # blockBufferSize reflects the size of the re-ordering buffer
                # which captures blocks and takes care to deliver them in order
                # down to the ledger layer. The actually buffer size is bounded
between
                # 0 and 2*blockBufferSize, each channel maintains its own buffer
                blockBufferSize: 100
                # maxRetries maximum number of re-tries to ask
                # for single state transfer request
                maxRetries: 3

        # TLS Settings
        tls:
            # Require server-side TLS
            enabled:  false
            # Require client certificates / mutual TLS.
            # Note that clients that are not configured to use a certificate will
            # fail to connect to the peer.
            clientAuthRequired: false
            # X.509 certificate used for TLS server
            cert:
                file: tls/server.crt
            # Private key used for TLS server (and client if clientAuthEnabled
            # is set to true
            key:
                file: tls/server.key
            # Trusted root certificate chain for tls.cert
            rootcert:
                file: tls/ca.crt
            # Set of root certificate authorities used to verify client
certificates
            clientRootCAs:
                files:
                  - tls/ca.crt
            # Private key used for TLS when making client connections.  If
            # not set, peer.tls.key.file will be used instead
            clientKey:
                file:
            # X.509 certificate used for TLS when making client connections.
            # If not set, peer.tls.cert.file will be used instead
            clientCert:
                file:

        # Authentication contains configuration parameters related to
authenticating
        # client messages
```

```
        authentication:
            # the acceptable difference between the current server time and the
            # client's time as specified in a client request message
            timewindow: 15m

        # Path on the file system where peer will store data (eg ledger). This
        # location must be access control protected to prevent unintended
        # modification that might corrupt the peer operations.
        fileSystemPath: /var/hyperledger/production

        # BCCSP (Blockchain crypto provider): Select which crypto implementation
or
        # library to use
        BCCSP:
            Default: SW
            # Settings for the SW crypto provider (i.e. when DEFAULT: SW)
            SW:
                # TODO: The default Hash and Security level needs refactoring to
be
                # fully configurable. Changing these defaults requires
coordination
                # SHA2 is hardcoded in several places, not only BCCSP
                Hash: SHA2
                Security: 256
                # Location of Key Store
                FileKeyStore:
                    # If "", defaults to 'mspConfigPath'/keystore
                    KeyStore:
            # Settings for the PKCS#11 crypto provider (i.e. when DEFAULT:
PKCS11)
            PKCS11:
                # Location of the PKCS11 module library
                Library:
                # Token Label
                Label:
                # User PIN
                Pin:
                Hash:
                Security:

        # Path on the file system where peer will find MSP local configurations
        mspConfigPath: msp

        # Identifier of the local MSP
        # ----!!!!IMPORTANT!!!-!!!IMPORTANT!!!-!!!IMPORTANT!!!!----
        # Deployers need to change the value of the localMspId string.
        # In particular, the name of the local MSP ID of a peer needs
        # to match the name of one of the MSPs in each of the channel
        # that this peer is a member of. Otherwise this peer's messages
        # will not be identified as valid by other nodes.
        localMspId: SampleOrg

        # CLI common client config options
        client:
            # connection timeout
            connTimeout: 3s

        # Delivery service related config
        deliveryclient:
```

```
            # It sets the total time the delivery service may spend in
reconnection
            # attempts until its retry logic gives up and returns an error
            reconnectTotalTimeThreshold: 3600s

            # It sets the delivery service <-> ordering service node connection
timeout
            connTimeout: 3s

            # It sets the delivery service maximal delay between consecutive
retries
            reConnectBackoffThreshold: 3600s

            # A list of orderer endpoint addresses which should be overridden
            # when found in channel configurations.
            addressOverrides:
            #  - from:
            #    to:
            #    caCertsFile:
            #  - from:
            #    to:
            #    caCertsFile:

        # Type for the local MSP - by default it's of type bccsp
        localMspType: bccsp

        # Used with Go profiling tools only in none production environment. In
        # production, it should be disabled (eg enabled: false)
        profile:
            enabled:     false
            listenAddress: 0.0.0.0:6060

        # Handlers defines custom handlers that can filter and mutate
        # objects passing within the peer, such as:
        #   Auth filter - reject or forward proposals from clients
        #   Decorators  - append or mutate the chaincode input passed to the
chaincode
        #   Endorsers   - Custom signing over proposal response payload and its
mutation
        # Valid handler definition contains:
        #   - A name which is a factory method name defined in
        #     core/handlers/library/library.go for statically compiled handlers
        #   - library path to shared object binary for pluggable filters
        # Auth filters and decorators are chained and executed in the order that
        # they are defined. For example:
        # authFilters:
        #   -
        #     name: FilterOne
        #     library: /opt/lib/filter.so
        #   -
        #     name: FilterTwo
        # decorators:
        #   -
        #     name: DecoratorOne
        #   -
        #     name: DecoratorTwo
        #     library: /opt/lib/decorator.so
        # Endorsers are configured as a map that its keys are the endorsement
system chaincodes that are being overridden.
```

```
        # Below is an example that overrides the default ESCC and uses an
endorsement plugin that has the same functionality
        # as the default ESCC.
        # If the 'library' property is missing, the name is used as the
constructor method in the builtin library similar
        # to auth filters and decorators.
        # endorsers:
        #   escc:
        #     name: DefaultESCC
        #     library: /etc/hyperledger/fabric/plugin/escc.so
        handlers:
            authFilters:
              -
                name: DefaultAuth
              -
                name: ExpirationCheck    # This filter checks identity x509
certificate expiration
            decorators:
              -
                name: DefaultDecorator
            endorsers:
              escc:
                name: DefaultEndorsement
                library:
            validators:
              vscc:
                name: DefaultValidation
                library:

        #    library: /etc/hyperledger/fabric/plugin/escc.so
        # Number of goroutines that will execute transaction validation in
parallel.
        # By default, the peer chooses the number of CPUs on the machine. Set
this
        # variable to override that choice.
        # NOTE: overriding this value might negatively influence the performance
of
        # the peer so please change this value only if you know what you're doing
        validatorPoolSize:

        # The discovery service is used by clients to query information about
peers,
        # such as - which peers have joined a certain channel, what is the latest
        # channel config, and most importantly - given a chaincode and a channel,
        # what possible sets of peers satisfy the endorsement policy.
        discovery:
            enabled: true
            # Whether the authentication cache is enabled or not.
            authCacheEnabled: true
            # The maximum size of the cache, after which a purge takes place
            authCacheMaxSize: 1000
            # The proportion (0 to 1) of entries that remain in the cache after
the cache is purged due to overpopulation
            authCachePurgeRetentionRatio: 0.75
            # Whether to allow non-admins to perform non channel scoped queries.
            # When this is false, it means that only peer admins can perform non
channel scoped queries.
            orgMembersAllowedAccess: false
```

```
        # Limits is used to configure some internal resource limits.
        limits:
          # Concurrency limits the number of concurrently running system
chaincode requests.
          # This option is only supported for qscc at this time.
          concurrency:
            qscc: 5000


###############################################################################
    #
    #    VM section
    #

###############################################################################
    vm:

        # Endpoint of the vm management system.  For docker can be one of the
following in general
        # unix:///var/run/docker.sock
        # http://localhost:2375
        # https://localhost:2376
        endpoint: unix:///var/run/docker.sock

        # settings for docker vms
        docker:
            tls:
                enabled: false
                ca:
                    file: docker/ca.crt
                cert:
                    file: docker/tls.crt
                key:
                    file: docker/tls.key

            # Enables/disables the standard out/err from chaincode containers for
            # debugging purposes
            attachStdout: false

            # Parameters on creating docker container.
            # Container may be efficiently created using ipam & dns-server for
cluster
            # NetworkMode - sets the networking mode for the container. Supported
            # standard values are: `host`(default),`bridge`,`ipvlan`,`none`.
            # Dns - a list of DNS servers for the container to use.
            # Note:  `Privileged` `Binds` `Links` and `PortBindings` properties
of
            # Docker Host Config are not supported and will not be used if set.
            # LogConfig - sets the logging driver (Type) and related options
            # (Config) for Docker. For more info,
            # https://docs.docker.com/engine/admin/logging/overview/
            # Note: Set LogConfig using Environment Variables is not supported.
            hostConfig:
                NetworkMode: host
                Dns:
                  # - 192.168.0.1
                LogConfig:
                    Type: json-file
                    Config:
```

```
                        max-size: "50m"
                        max-file: "5"
                Memory: 2147483648


###############################################################################
    #
    #    Chaincode section
    #

###############################################################################
    chaincode:

        # The id is used by the Chaincode stub to register the executing
Chaincode
        # ID with the Peer and is generally supplied through ENV variables
        # the `path` form of ID is provided when installing the chaincode.
        # The `name` is used for all other requests and can be any string.
        id:
            path:
            name:

        # Generic builder environment, suitable for most chaincode types
        builder: $(DOCKER_NS)/fabric-ccenv:$(PROJECT_VERSION)

        # Enables/disables force pulling of the base docker images (listed below)
        # during user chaincode instantiation.
        # Useful when using moving image tags (such as :latest)
        pull: false

        golang:
            # golang will never need more than baseos
            runtime: $(DOCKER_NS)/fabric-baseos:$(PROJECT_VERSION)

            # whether or not golang chaincode should be linked dynamically
            dynamicLink: false

        java:
            # This is an image based on java:openjdk-8 with addition compiler
            # tools added for java shim layer packaging.
            # This image is packed with shim layer libraries that are necessary
            # for Java chaincode runtime.
            runtime: $(DOCKER_NS)/fabric-javaenv:latest

        node:
            # This is an image based on node:$(NODE_VER)-alpine
            runtime: $(DOCKER_NS)/fabric-nodeenv:latest

        # List of directories to treat as external builders and launchers for
        # chaincode. The external builder detection processing will iterate over
the
        # builders in the order specified below.
        externalBuilders:
          - name: mygolangbuilder
            path: /builders/golang/
            environmentWhitelist:
               - GOPROXY
```

```
        # The maximum duration to wait for the chaincode build and install
process
        # to complete.
        installTimeout: 300s

        # Timeout duration for starting up a container and waiting for Register
        # to come through.
        startuptimeout: 300s

        # Timeout duration for Invoke and Init calls to prevent runaway.
        # This timeout is used by all chaincodes in all the channels, including
        # system chaincodes.
        # Note that during Invoke, if the image is not available (e.g. being
        # cleaned up when in development environment), the peer will
automatically
        # build the image, which might take more time. In production environment,
        # the chaincode image is unlikely to be deleted, so the timeout could be
        # reduced accordingly.
        executetimeout: 30s

        # There are 2 modes: "dev" and "net".
        # In dev mode, user runs the chaincode after starting peer from
        # command line on local machine.
        # In net mode, peer will run chaincode in a docker container.
        mode: net

        # keepalive in seconds. In situations where the communiction goes through
a
        # proxy that does not support keep-alive, this parameter will maintain
connection
        # between peer and chaincode.
        # A value <= 0 turns keepalive off
        keepalive: 0

        # system chaincodes whitelist. To add system chaincode "myscc" to the
        # whitelist, add "myscc: enable" to the list below, and register in
        # chaincode/importsysccs.go
        system:
            _lifecycle: enable
            cscc: enable
            lscc: enable
            escc: enable
            vscc: enable
            qscc: enable

        # Logging section for the chaincode container
        logging:
          # Default level for all loggers within the chaincode container
          level:  info
          # Override default level for the 'shim' logger
          shim:   warning
          # Format for the chaincode container logs
          format: '%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}]
%{shortfunc} -> %{level:.4s} %{id:03x}%{color:reset} %{message}'


###############################################################################
    #
    #    Ledger section - ledger configuration encompasses both the blockchain
```

```
    #    and the state
    #

###############################################################################
    ledger:

      blockchain:

      state:
        # stateDatabase - options are "goleveldb", "CouchDB"
        # goleveldb - default state database stored in goleveldb.
        # CouchDB - store state database in CouchDB
        stateDatabase: goleveldb
        # Limit on the number of records to return per query
        totalQueryLimit: 100000
        couchDBConfig:
          # It is recommended to run CouchDB on the same server as the peer, and
          # not map the CouchDB container port to a server port in docker-
compose.
          # Otherwise proper security must be provided on the connection between
          # CouchDB client (on the peer) and server.
          couchDBAddress: 127.0.0.1:5984
          # This username must have read and write authority on CouchDB
          username:
          # The password is recommended to pass as an environment variable
          # during start up (eg CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD).
          # If it is stored here, the file must be access control protected
          # to prevent unintended users from discovering the password.
          password:
          # Number of retries for CouchDB errors
          maxRetries: 3
          # Number of retries for CouchDB errors during peer startup
          maxRetriesOnStartup: 12
          # CouchDB request timeout (unit: duration, e.g. 20s)
          requestTimeout: 35s
          # Limit on the number of records per each CouchDB query
          # Note that chaincode queries are only bound by totalQueryLimit.
          # Internally the chaincode may execute multiple CouchDB queries,
          # each of size internalQueryLimit.
          internalQueryLimit: 1000
          # Limit on the number of records per CouchDB bulk update batch
          maxBatchUpdateSize: 1000
          # Warm indexes after every N blocks.
          # This option warms any indexes that have been
          # deployed to CouchDB after every N blocks.
          # A value of 1 will warm indexes after every block commit,
          # to ensure fast selector queries.
          # Increasing the value may improve write efficiency of peer and
CouchDB,
          # but may degrade query response time.
          warmIndexesAfterNBlocks: 1
          # Create the _global_changes system database
          # This is optional.  Creating the global changes database will require
          # additional system resources to track changes and maintain the
database
          createGlobalChangesDB: false
          # CacheSize denotes the maximum mega bytes (MB) to be allocated for the
in-memory state
```

```
          # cache. Note that CacheSize needs to be a multiple of 32 MB. If it is
not a multiple
          # of 32 MB, the peer would round the size to the next multiple of 32
MB.
          # To disable the cache, 0 MB needs to be assigned to the cacheSize.
          cacheSize: 64

      history:
        # enableHistoryDatabase - options are true or false
        # Indicates if the history of key updates should be stored.
        # All history 'index' will be stored in goleveldb, regardless if using
        # CouchDB or alternate database for the state.
        enableHistoryDatabase: true

      pvtdataStore:
        # the maximum db batch size for converting
        # the ineligible missing data entries to eligible missing data entries
        collElgProcMaxDbBatchSize: 5000
        # the minimum duration (in milliseconds) between writing
        # two consecutive db batches for converting the ineligible missing data
entries to eligible missing data entries
        collElgProcDbBatchesInterval: 1000


###############################################################################
    #
    #    Operations section
    #

###############################################################################
    operations:
        # host and port for the operations server
        listenAddress: 127.0.0.1:9443

        # TLS configuration for the operations endpoint
        tls:
            # TLS enabled
            enabled: false

            # path to PEM encoded server certificate for the operations server
            cert:
                file:

            # path to PEM encoded server key for the operations server
            key:
                file:

            # most operations service endpoints require client authentication
when TLS
            # is enabled. clientAuthRequired requires client certificate
authentication
            # at the TLS layer to access all resources.
            clientAuthRequired: false

            # paths to PEM encoded ca certificates to trust for client
authentication
            clientRootCAs:
                files: []
```

```
###############################################################################
    #
    #    Metrics section
    #

###############################################################################
    metrics:
        # metrics provider is one of statsd, prometheus, or disabled
        provider: disabled

        # statsd configuration
        statsd:
            # network type: tcp or udp
            network: udp

            # statsd server address
            address: 127.0.0.1:8125

            # the interval at which locally cached counters and gauges are pushed
            # to statsd; timings are pushed immediately
            writeInterval: 10s

            # prefix is prepended to all emitted statsd metrics
            prefix:
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: peer0-org0alfa-pasbck-new-com
  name: peer0-org0alfa-pasbck-new-com
  namespace: new-bck
spec:
  type: ClusterIP
  ports:
  - name: "peer-core"
    port: 7051
    targetPort: 7051
  - name: "peer-chaincode-events"
    port: 7052
    targetPort: 7052
  - name: "7053"
    port: 7053
    targetPort: 7053
  - name: "7054"
    port: 7054
    targetPort: 7054
  selector:
    app: peer0-org0alfa-pasbck-new-com
---
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: peer0-org0alfa-pasbck-new-com
  name: peer0-org0alfa-pasbck-new-com
  namespace: new-bck
```

```
spec:
  selector:
    matchLabels:
      app: peer0-org0alfa-pasbck-new-com
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: peer0-org0alfa-pasbck-new-com
        logging.elk.stack: platform
    spec:
      containers:
      - args:
        - peer
        - node
        - start
        env:
        - name: FABRIC_LOGGING_SPEC
          value: DEBUG
        - name: CORE_PEER_TLS_ENABLED
          value: "true"
        - name: CORE_PEER_GOSSIP_USELEADERELECTION
          value: "true"
        - name: CORE_PEER_GOSSIP_ORGLEADER
          value: "false"
        - name: CORE_PEER_PROFILE_ENABLED
          value: "true"
        - name: CORE_PEER_MSPCONFIGPATH
          value: /etc/hyperledger/fabric/crypto-
config/peerOrganizations/org0alfa-pasbck-new-com/peers/peer0-org0alfa-pasbck-new-
com/msp/
        - name: CORE_PEER_TLS_CERT_FILE
          value: /etc/hyperledger/fabric/crypto-
config/peerOrganizations/org0alfa-pasbck-new-com/peers/peer0-org0alfa-pasbck-new-
com/tls/server.crt
        - name: CORE_PEER_TLS_KEY_FILE
          value: /etc/hyperledger/fabric/crypto-
config/peerOrganizations/org0alfa-pasbck-new-com/peers/peer0-org0alfa-pasbck-new-
com/tls/server.key
        - name: CORE_PEER_TLS_ROOTCERT_FILE
          value: /etc/hyperledger/fabric/crypto-
config/peerOrganizations/org0alfa-pasbck-new-com/peers/peer0-org0alfa-pasbck-new-
com/tls/ca.crt
        - name: CORE_CHAINCODE_EXECUTETIMEOUT  # Allow more time for chaincode
container to build on install.
          value: 300s
        - name: CORE_LEDGER_STATE_STATEDATABASE
          value: CouchDB
        - name: CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS
          value: "couchdb-org0alfa:5984"
        - name: CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME
          value: user0
        - name: CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD
          value: pass0
        - name: CORE_PEER_ADDRESS
          value: peer0-org0alfa-pasbck-new-com:7051
        - name: CORE_PEER_ID
```

```
          value: peer0-org0alfa-pasbck-new-com
        - name: CORE_PEER_LISTENADDRESS
          value: 0.0.0.0:7051
        - name: CORE_PEER_CHAINCODEADDRESS
          value: peer0-org0alfa-chaincode:7052
        - name: CORE_PEER_CHAINCODELISTENADDRESS
          value: "0.0.0.0:7052"
        - name: CORE_PEER_GOSSIP_EXTERNALENDPOINT
          value: peer0-org0alfa-pasbck-new-com:7051
        - name: CORE_PEER_LOCALMSPID
          value: Org0alfaMSP
        image: peer-tools:2.0.5
        imagePullPolicy: IfNotPresent
        resources:
            requests:
              memory: "64Mi"
              cpu: "50m"
            limits:
              memory: "128Mi"
              cpu: "100m"
        name: peer0-org0alfa-pasbck-new
        ports:
        - containerPort: 7051
        - containerPort: 7052
        - containerPort: 7053
        - containerPort: 7054
        volumeMounts:
        - mountPath: /var/hyperledger/production
          name: peer0org0alfa-persistentdata
        - mountPath: /etc/hyperledger/fabric/
          name: peer0-org0alfa-cli-crypto
        - mountPath: /etc/hyperledger/fabric/core.yaml
          name: builders-config
          subPath: core.yaml
        workingDir: /opt/gopath/src/github.com/hyperledger/fabric/peer
      restartPolicy: Always
      volumes:
      - name: peer0org0alfa-persistentdata
        persistentVolumeClaim:
          claimName: pasbck-peer0alfa-new
      - name: peer0-org0alfa-cli-crypto
        persistentVolumeClaim:
          claimName: pasbck-cli-new
      - name: builders-config
        configMap:
          name: builders-config
          items:
            - key: core.yaml
              path: core.yaml
---
```

Orderer0

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: ordererb-pasbck-new-com
  name: ordererb-pasbck-new-com
  namespace: new-bck
spec:
```

```
  type: ClusterIP
  ports:
  - name: "orderer"
    port: 7050
    targetPort: 7050
  selector:
    app: ordererb-pasbck-new-com
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: ordererb-pasbck-new-com
    metrics-service: "true"
  name: ordererb-metrics
  namespace: new-bck
spec:
  type: ClusterIP
  ports:
  - name: "orderer-metrics"
    port: 8443
    targetPort: 8443
  selector:
    app: ordererb-pasbck-new-com
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: ordererb-pasbck-new-com
  name: ordererb-pasbck-new-com
  namespace: new-bck
spec:
  selector:
    matchLabels:
      app: ordererb-pasbck-new-com
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: ordererb-pasbck-new-com
        logging.elk.stack: platform
    spec:
      containers:
      - args:
        - orderer
        env:
        - name: FABRIC_LOGGING_SPEC
          value: INFO
        - name: ORDERER_GENERAL_GENESISFILE
          value: /var/hyperledger/channel-artifacts/genesis.block
#/var/hyperledger/orderer/genesis.block
        - name: ORDERER_GENERAL_GENESISMETHOD
          value: file
        - name: ORDERER_GENERAL_LISTENADDRESS
          value: 0.0.0.0
        - name: ORDERER_GENERAL_LOCALMSPDIR
```

```
          value: /var/hyperledger/crypto-config/ordererOrganizations/pasbck-new-
com/orderers/ordererb-pasbck-new-com/msp #/var/hyperledger/orderer/msp
        - name: ORDERER_GENERAL_LOCALMSPID
          value: OrdererMSP
        - name: ORDERER_GENERAL_LOGLEVEL
          value: debug
        - name: ORDERER_GENERAL_TLS_CERTIFICATE
          value: /var/hyperledger/crypto-config/ordererOrganizations/pasbck-new-
com/orderers/ordererb-pasbck-new-com/tls/server.crt
#/var/hyperledger/orderer/tls/server.crt
        - name: ORDERER_GENERAL_TLS_ENABLED
          value: "true"
        - name: ORDERER_GENERAL_TLS_PRIVATEKEY
          value: /var/hyperledger/crypto-config/ordererOrganizations/pasbck-new-
com/orderers/ordererb-pasbck-new-com/tls/server.key
#/var/hyperledger/orderer/tls/server.key
        - name: ORDERER_GENERAL_TLS_ROOTCAS
          value: "[/var/hyperledger/crypto-config/ordererOrganizations/pasbck-
new-com/orderers/ordererb-pasbck-new-com/tls/ca.crt]"
        - name: ORDERER_GENERAL_CLUSTER_CLIENTCERTIFICATE
          value: /var/hyperledger/crypto-config/ordererOrganizations/pasbck-new-
com/orderers/ordererb-pasbck-new-com/tls/server.crt
        - name: ORDERER_GENERAL_CLUSTER_CLIENTPRIVATEKEY
          value: /var/hyperledger/crypto-config/ordererOrganizations/pasbck-new-
com/orderers/ordererb-pasbck-new-com/tls/server.key
        - name: ORDERER_GENERAL_CLUSTER_ROOTCAS
          value: "[/var/hyperledger/crypto-config/ordererOrganizations/pasbck-
new-com/orderers/ordererb-pasbck-new-com/tls/ca.crt]"
        - name: ORDERER_METRICS_PROVIDER
          value: prometheus
        - name: ORDERER_OPERATIONS_LISTENADDRESS
          value: 0.0.0.0:8443
        image: fabric-orderer:2.0.1
        imagePullPolicy: IfNotPresent
        resources:
          requests:
            memory: "64Mi"
            cpu: "50m"
          limits:
            memory: "128Mi"
            cpu: "100m"
        name: ordererb-pasbck-new-com
        ports:
        - containerPort: 7050
        - containerPort: 8443
        volumeMounts:
        - mountPath: /var/hyperledger/production
          name: ordererb-persistentdata
        - mountPath: /var/hyperledger/
          name: ordererb-cli-config-files
        workingDir: /opt/gopath/src/github.com/hyperledger/fabric
      restartPolicy: Always
      volumes:
      - name: ordererb-persistentdata
        persistentVolumeClaim:
          claimName: pasbck-ordererb-new
      - name: ordererb-cli-config-files
        persistentVolumeClaim:
          claimName: pasbck-cli-new
```

chaincode0A

```
#---------------- Chaincode Service ---------------------
apiVersion: v1
kind: Service
metadata:
  name: peer0-org0alfa-chaincode
  namespace: new-bck
  labels:
    app: peer0-org0alfa-chaincode
spec:
  ports:
    - name: grpc
      port: 7052
      targetPort: 7052
  selector:
    app: peer0-org0alfa-chaincode
---
#---------------- Chaincode Deployment --------------------
apiVersion: apps/v1 # for versions before 1.8.0 use apps/v1beta1
kind: Deployment
metadata:
  name: peer0-org0alfa-chaincode
  namespace: new-bck
  labels:
    app: peer0-org0alfa-chaincode
spec:
  selector:
    matchLabels:
      app: peer0-org0alfa-chaincode
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: peer0-org0alfa-chaincode
        logging.elk.stack: platform
    spec:
      containers:
        - image: cc-image-cc:7   #chaincode/redhat-chaincode:2
          name: peer0-org0alfa-chaincode
          imagePullPolicy: IfNotPresent
          env:
            - name: CHAINCODE_CCID
              value:
"addendum_v1:1528162346496c7dec8637bf769a1f365e84dc14b38ed4f27363ca41db3ea37d"
            - name: CHAINCODE_ADDRESS
              value: "0.0.0.0:7052"
          resources:
            requests:
              memory: "64Mi"
              cpu: "125m"
            limits:
              memory: "128Mi"
              cpu: "250m"
          ports:
            - containerPort: 7052
```

Ca-cetificate0A

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: ca-org0alfa
  name: ca-org0alfa
  namespace: new-bck
spec:
  type: ClusterIP
  ports:
  - name: "ca-org0alfa"
    port: 7054
    targetPort: 7054
  selector:
    app: ca-org0alfa
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: ca-org0alfa
  name: ca-org0alfa
  namespace: new-bck
spec:
  selector:
    matchLabels:
      app: ca-org0alfa
  replicas: 1
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: ca-org0alfa
        logging.elk.stack: platform
    spec:
      containers:
      - args:
        - sh
        - -c
        - fabric-ca-server start --ca.certfile /etc/hyperledger/fabric-ca-server-
config/crypto-config/peerOrganizations/org0alfa-pasbck-new-com/ca/ca.org0alfa-
pasbck-new-com-cert.pem
          --ca.keyfile /etc/hyperledger/fabric-ca-server-config/crypto-
config/peerOrganizations/org0alfa-pasbck-new-com/ca/priv_sk
          -b admin:adminpw -d
        env:
        - name: FABRIC_CA_HOME
          value: /etc/hyperledger/fabric-ca-server
        - name: FABRIC_CA_SERVER_CA_NAME
          value: ca-org0alfa
        - name: FABRIC_CA_SERVER_TLS_CERTFILE
          value: /etc/hyperledger/fabric-ca-server-config/crypto-
config/peerOrganizations/org0alfa-pasbck-new-com/ca/ca.org0alfa-pasbck-new-com-
cert.pem
        - name: FABRIC_CA_SERVER_TLS_ENABLED
          value: "true"
        - name: FABRIC_CA_SERVER_TLS_KEYFILE
```

```
          value: /etc/hyperledger/fabric-ca-server-config/crypto-
config/peerOrganizations/org0alfa-pasbck-new-com/ca/priv_sk
        - name: FABRIC_CA_SERVER_PORT
          value: "7054"
        resources:
          requests:
            memory: "64Mi"
            cpu: "50m"
          limits:
            memory: "128Mi"
            cpu: "100m"
        image: ca-tools:2.0.1
        name: ca-org0alfa
        ports:
        - containerPort: 7054
        volumeMounts:
        - mountPath: /etc/hyperledger/fabric-ca-server-config
          name: ca-org0alfa-claim0
        - mountPath: /etc/hyperledger/fabric-ca-server
          name: ca-org0alfa-claim1
      restartPolicy: Always
      volumes:
      - name: ca-org0alfa-claim0
        persistentVolumeClaim:
          claimName: pasbck-cli-new
      - name: ca-org0alfa-claim1
        persistentVolumeClaim:
          claimName: pasbck-ca0alfa-new
```

Anexo 6. Docker-in-Docker Dockerfile

```
FROM docker:19.03-dind

# busybox "ip" is insufficient:
#   [rootlesskit:child ] error: executing [[ip tuntap add name tap0 mode tap] [ip
link set tap0 address 02:50:00:00:00:01]]: exit status 1
RUN apk add --no-cache iproute2

# "/run/user/UID" will be used by default as the value of XDG_RUNTIME_DIR
RUN mkdir /run/user && chmod 1777 /run/user

# create a default user preconfigured for running rootless dockerd
RUN set -eux; \
       adduser -h /home/rootless -g 'Rootless' -D -u 1000 rootless; \
       echo 'rootless:100000:65536' >> /etc/subuid; \
       echo 'rootless:100000:65536' >> /etc/subgid

RUN set -eux; \
       \
# this "case" statement is generated via "update.sh"
       apkArch="$(apk --print-arch)"; \
       case "$apkArch" in \
# amd64
            x86_64) dockerArch='x86_64' ;; \
# arm32v6
            armhf) dockerArch='armel' ;; \
# arm32v7
```

```
            armv7) dockerArch='armhf' ;; \
# arm64v8
            aarch64) dockerArch='aarch64' ;; \
            *) echo >&2 "error: unsupported architecture ($apkArch)"; exit 1 ;;\
      esac; \
      \
      if ! wget -O rootless.tgz
"https://download.docker.com/linux/static/${DOCKER_CHANNEL}/${dockerArch}/docker-
rootless-extras-${DOCKER_VERSION}.tgz"; then \
            echo >&2 "error: failed to download 'docker-rootless-extras-
${DOCKER_VERSION}' from '${DOCKER_CHANNEL}' for '${dockerArch}'"; \
            exit 1; \
      fi; \
      \
      tar --extract \
            --file rootless.tgz \
            --strip-components 1 \
            --directory /usr/local/bin/ \
            'docker-rootless-extras/vpnkit' \
      ; \
      rm rootless.tgz; \
      \
# we download/build rootlesskit separately to get a newer release
#     rootlesskit --version; \
      vpnkit --version

# https://github.com/rootless-containers/rootlesskit/releases
ENV ROOTLESSKIT_VERSION 0.7.0

RUN set -eux; \
      apk add --no-cache --virtual .rootlesskit-build-deps \
            go \
            libc-dev \
      ; \
      wget -O rootlesskit.tgz "https://github.com/rootless-
containers/rootlesskit/archive/v${ROOTLESSKIT_VERSION}.tar.gz"; \
      export GOPATH='/go'; mkdir "$GOPATH"; \
      mkdir -p "$GOPATH/src/github.com/rootless-containers/rootlesskit"; \
      tar --extract --file rootlesskit.tgz --directory
"$GOPATH/src/github.com/rootless-containers/rootlesskit" --strip-components 1; \
      rm rootlesskit.tgz; \
      go build -o /usr/local/bin/rootlesskit github.com/rootless-
containers/rootlesskit/cmd/rootlesskit; \
      go build -o /usr/local/bin/rootlesskit-docker-proxy github.com/rootless-
containers/rootlesskit/cmd/rootlesskit-docker-proxy; \
      rm -rf "$GOPATH"; \
      apk del --no-network .rootlesskit-build-deps; \
      rootlesskit --version

# pre-create "/var/lib/docker" for our rootless user
RUN set -eux; \
      mkdir -p /home/rootless/.local/share/docker; \
      chown -R rootless:rootless /home/rootless/.local/share/docker
VOLUME /home/rootless/.local/share/docker
USER rootless
```

Anexo 7. Dockerfile imagen rootless producción

Dockerfile imagen rootless peer Hyperledger fabric V2.0.1

```
FROM hyperledger/fabric-peer:amd64-2.0.1
RUN addgroup --gid 1022 chaincode && adduser --uid 1022 --disabled-password --
ingroup chaincode chaincode && \
    chown -R 1022:1022 /var/hyperledger/* && \
    chown -R 1022:1022 /var/run/* && \
    chmod 744 /var/hyperledger && \
    mkdir -p /opt/gopath/src/github.com/hyperledger/fabric/peer/ && \
    chown -R 1022:1022 /opt/* && \
    chown -R 1022:1022 /etc/hyperledger/*
    mkdir -p
USER 1022
```

Dockerfile imagen rootless orderer Hyperledger fabric V2.0.1

```
FROM hyperledger/fabric-orderer:amd64-2.0.1
RUN addgroup --gid 1022 chaincode && adduser --uid 1022 --disabled-password --
ingroup chaincode chaincode && \
    chown -R 1022:1022  /var/ && \
    mkdir -p /opt/gopath/src/github.com/hyperledger/fabric && \
    chown -R 1022:1022 /opt/gopath/
USER 1022
```

Dockerfile imagen rootless cli Hyperledger fabric V2.0.1

```
FROM hyperledger/fabric-tools:amd64-2.0.1
RUN mkdir -p /opt/gopath/src/github.com/hyperledger/fabric/peer
COPY configtx.yaml /opt/gopath/src/github.com/hyperledger/fabric
COPY crypto-config.yaml /opt/gopath/src/github.com/hyperledger/fabric
RUN addgroup --gid 1022 chaincode && adduser --uid 1022 --disabled-password --
ingroup chaincode chaincode
RUN chown -R 1022:chaincode /opt/gopath/src/github.com/hyperledger/fabric && \
    mkdir -p /etc/hyperledger/fabric/ && \
    chmod +x /opt/gopath/src/github.com/hyperledger/fabric/crypto-config.yaml &&
\
    chown -R 1022:chaincode /etc/hyperledger/fabric/ && \
    chown -R 1022:chaincode /etc/hyperledger/fabric/configtx.yaml && \
    chown -R 1022:chaincode /etc/hyperledger/fabric/core.yaml && \
    chown -R 1022:chaincode /etc/hyperledger/fabric/msp/ && \
    chown -R 1022:chaincode /etc/hyperledger/fabric/orderer.yaml
USER 1022
```

Dockerfile imagen rootless ca Hyperledger fabric V2.0.1

```
FROM hyperledger/fabric-ca:1.4
RUN addgroup --gid 1022 chaincode && adduser --uid 1022 --disabled-password  --
no-create-home --ingroup chaincode chaincode  && \
    chown -R 1022:1022 /etc/hyperledger/*
USER 1022
```

Dockerfile imagen rootless chaincode Hyperledger fabric V2.0.1

```
# This image is a microservice in golang for the Degree chaincode
FROM registry.redhat.io/rhel8/go-toolset:1.12.8  AS build

COPY ./ /go/src/github.com/addendum
WORKDIR /go/src/github.com/addendum

# Build application

RUN scl enable go-toolset-1.12.8 'go mod init github.com/addendum'
RUN go mod init github.com/addendum
RUN scl enable go-toolset-1.12.8 'go build -o chaincode -v .'
RUN go build -o chaincode -v .

# Production ready image
# Pass the binary to the prod image
FROM registry.redhat.io/rhel8/go-toolset:1.12.8 as prod
COPY --from=build /go/src/github.com/addendum/chaincode /app/chaincode
USER 1022
WORKDIR /app
CMD ./chaincode
```

Dockerfile imagen rootless couchDB Hyperledger fabric V2.0.1

```
FROM hyperledger/fabric-couchdb:amd64-0.4
COPY ./docker-entrypoint.sh /
RUN chmod +x /docker-entrypoint.sh \
 && chown -R couchdb:couchdb /opt/couchdb/
ENTRYPOINT ["tini", "--", "/docker-entrypoint.sh"]
CMD ["/opt/couchdb/bin/couchdb"]
USER couchdb
```

Anexo 8. Configuración Hyperledger Fabric

configtx.yaml

```
# Copyright IBM Corp. All Rights Reserved.
#
# SPDX-License-Identifier: Apache-2.0
#

---
################################################################################
#
#   Section: Organizations
#
#   - This section defines the different organizational identities which will
#   be referenced later in the configuration.
#
################################################################################
Organizations:

   # SampleOrg defines an MSP using the sampleconfig.  It should never be used
   # in production but may be used as a template for other definitions
   - &OrdererOrg
```

```
        # DefaultOrg defines the organization which is used in the sampleconfig
        # of the fabric.git development environment
        Name: OrdererOrg

        # ID to load the MSP definition as
        ID: OrdererMSP

        # MSPDir is the filesystem path which contains the MSP configuration
        MSPDir: crypto-config/ordererOrganizations/pasbck-new-com/msp

        # Policies defines the set of policies at this level of the config tree
        # For organization policies, their canonical path is usually
        #   /Channel/<Application|Orderer>/<OrgName>/<PolicyName>
        Policies: &ordererPolicies
            Readers:
                Type: Signature
                Rule: "OR('OrdererMSP.member')"
            Writers:
                Type: Signature
                Rule: "OR('OrdererMSP.member')"
            Admins:
                Type: Signature
                Rule: "OR('OrdererMSP.admin')"

    - &Org0alfa
        # DefaultOrg defines the organization which is used in the sampleconfig
        # of the fabric.git development environment
        Name: Org0alfaMSP

        # ID to load the MSP definition as
        ID: Org0alfaMSP

        MSPDir: crypto-config/peerOrganizations/org0alfa-pasbck-new-com/msp

        # Policies defines the set of policies at this level of the config tree
        # For organization policies, their canonical path is usually
        #   /Channel/<Application|Orderer>/<OrgName>/<PolicyName>
        Policies: &Org0alfaPolicies
            Readers:
                Type: Signature
                Rule: "OR('Org0alfaMSP.admin', 'Org0alfaMSP.peer',
'Org0alfaMSP.client')"
            Writers:
                Type: Signature
                Rule: "OR('Org0alfaMSP.admin', 'Org0alfaMSP.client')"
            Admins:
                Type: Signature
                Rule: "OR('Org0alfaMSP.admin')"
            Endorsement:
                Type: Signature
                Rule: "OR('Org0alfaMSP.peer')"

        AnchorPeers:
            # AnchorPeers defines the location of peers which can be used
            # for cross org gossip communication.  Note, this value is only
            # encoded in the genesis block in the Application section context
            - Host: peer0-org0alfa-pasbck-new-com
              Port: 7051
            # - Host: peer1-org0alfa
```

```
            #   Port: 7051

    - &Org0beta
        # DefaultOrg defines the organization which is used in the sampleconfig
        # of the fabric.git development environment
        Name: Org0betaMSP

        # ID to load the MSP definition as
        ID: Org0betaMSP

        MSPDir: crypto-config/peerOrganizations/org0beta-pasbck-new-com/msp

        # Policies defines the set of policies at this level of the config tree
        # For organization policies, their canonical path is usually
        #   /Channel/<Application|Orderer>/<OrgName>/<PolicyName>
        Policies: &Org0betaPolicies
            Readers:
                Type: Signature
                Rule: "OR('Org0betaMSP.admin', 'Org0betaMSP.peer',
'Org0betaMSP.client')"
            Writers:
                Type: Signature
                Rule: "OR('Org0betaMSP.admin', 'Org0betaMSP.client')"
            Admins:
                Type: Signature
                Rule: "OR('Org0betaMSP.admin')"
            Endorsement:
                Type: Signature
                Rule: "OR('Org0betaMSP.peer')"

        AnchorPeers:
            # AnchorPeers defines the location of peers which can be used
            # for cross org gossip communication.  Note, this value is only
            # encoded in the genesis block in the Application section context
            - Host: peer0-org0beta-pasbck-new-com
              Port: 7051
            # - Host: peer1-org0beta
            #   Port: 7051

################################################################################
#
#   SECTION: Capabilities
#
#   - This section defines the capabilities of fabric network. This is a new
#   concept as of v1.1.0 and should not be utilized in mixed networks with
#   v1.0.x peers and orderers.  Capabilities define features which must be
#   present in a fabric binary for that binary to safely participate in the
#   fabric network.  For instance, if a new MSP type is added, newer binaries
#   might recognize and validate the signatures from this type, while older
#   binaries without this support would be unable to validate those
#   transactions.  This could lead to different versions of the fabric binaries
#   having different world states.  Instead, defining a capability for a channel
#   informs those binaries without this capability that they must cease
#   processing transactions until they have been upgraded.  For v1.0.x if any
#   capabilities are defined (including a map with all capabilities turned off)
#   then the v1.0.x peer will deliberately crash.
#
################################################################################
Capabilities:
```

```
   # Channel capabilities apply to both the orderers and the peers and must be
   # supported by both.
   # Set the value of the capability to true to require it.
   Channel: &ChannelCapabilities
       # V2.0 for Channel is a catchall flag for behavior which has been
       # determined to be desired for all orderers and peers running at the
v2.0.0
       # level, but which would be incompatible with orderers and peers from
       # prior releases.
       # Prior to enabling V2.0 channel capabilities, ensure that all
       # orderers and peers on a channel are at v2.0.0 or later.
       V2_0: true

   # Orderer capabilities apply only to the orderers, and may be safely
   # used with prior release peers.
   # Set the value of the capability to true to require it.
   Orderer: &OrdererCapabilities
       # V1.1 for Orderer is a catchall flag for behavior which has been
       # determined to be desired for all orderers running at the v1.1.x
       # level, but which would be incompatible with orderers from prior
releases.
       # Prior to enabling V2.0 orderer capabilities, ensure that all
       # orderers on a channel are at v2.0.0 or later.
       V2_0: true

   # Application capabilities apply only to the peer network, and may be safely
   # used with prior release orderers.
   # Set the value of the capability to true to require it.
   Application: &ApplicationCapabilities
       # V2.0 for Application enables the new non-backwards compatible
       # features and fixes of fabric v2.0.
       # Prior to enabling V2.0 orderer capabilities, ensure that all
       # orderers on a channel are at v2.0.0 or later.
       V2_0: true

################################################################################
#
#   SECTION: Application
#
#   - This section defines the values to encode into a config transaction or
#   genesis block for application related parameters
#
################################################################################
Application: &ApplicationDefaults
   # ACLs: &ACLsDefault
   #     # This section provides defaults for policies for various resources
   #     # in the system. These "resources" could be functions on system
chaincodes
   #     # (e.g., "GetBlockByNumber" on the "qscc" system chaincode) or other
resources
   #     # (e.g.,who can receive Block events). This section does NOT specify the
resource's
   #     # definition or API, but just the ACL policy for it.
   #     #
   #     # User's can override these defaults with their own policy mapping by
defining the
   #     # mapping under ACLs in their channel definition
```

```
   #     #---New Lifecycle System Chaincode (_lifecycle) function to policy
mapping for access control--#

   #     # ACL policy for _lifecycle's "CommitChaincodeDefinition" function
   #     _lifecycle/CommitChaincodeDefinition: /Channel/Application/Writers

   #     # ACL policy for _lifecycle's "QueryChaincodeDefinition" function
   #     _lifecycle/QueryChaincodeDefinition: /Channel/Application/Readers

   #     # ACL policy for _lifecycle's "QueryNamespaceDefinitions" function
   #     _lifecycle/QueryNamespaceDefinitions: /Channel/Application/Readers

   #     #---Lifecycle System Chaincode (lscc) function to policy mapping for
access control---#

   #     # ACL policy for lscc's "getid" function
   #     lscc/ChaincodeExists: /Channel/Application/Readers

   #     # ACL policy for lscc's "getdepspec" function
   #     lscc/GetDeploymentSpec: /Channel/Application/Readers

   #     # ACL policy for lscc's "getccdata" function
   #     lscc/GetChaincodeData: /Channel/Application/Readers

   #     # ACL Policy for lscc's "getchaincodes" function
   #     lscc/GetInstantiatedChaincodes: /Channel/Application/Readers

   #     #---Query System Chaincode (qscc) function to policy mapping for access
control---#

   #     # ACL policy for qscc's "GetChainInfo" function
   #     qscc/GetChainInfo: /Channel/Application/Readers

   #     # ACL policy for qscc's "GetBlockByNumber" function
   #     qscc/GetBlockByNumber: /Channel/Application/Readers

   #     # ACL policy for qscc's  "GetBlockByHash" function
   #     qscc/GetBlockByHash: /Channel/Application/Readers

   #     # ACL policy for qscc's "GetTransactionByID" function
   #     qscc/GetTransactionByID: /Channel/Application/Readers

   #     # ACL policy for qscc's "GetBlockByTxID" function
   #     qscc/GetBlockByTxID: /Channel/Application/Readers

   #     #---Configuration System Chaincode (cscc) function to policy mapping for
access control---#

   #     # ACL policy for cscc's "GetConfigBlock" function
   #     cscc/GetConfigBlock: /Channel/Application/Readers

   #     # ACL policy for cscc's "GetConfigTree" function
   #     cscc/GetConfigTree: /Channel/Application/Readers

   #     # ACL policy for cscc's "SimulateConfigTreeUpdate" function
   #     cscc/SimulateConfigTreeUpdate: /Channel/Application/Readers

   #     #---Miscellanesous peer function to policy mapping for access control---
#
```

```
    #     # ACL policy for invoking chaincodes on peer
    #     peer/Propose: /Channel/Application/Writers

    #     # ACL policy for chaincode to chaincode invocation
    #     peer/ChaincodeToChaincode: /Channel/Application/Readers

    #     #---Events resource to policy mapping for access control###---#

    #     # ACL policy for sending block events
    #     event/Block: /Channel/Application/Readers

    #     # ACL policy for sending filtered block events
    #     event/FilteredBlock: /Channel/Application/Readers

    # Organizations is the list of orgs which are defined as participants on
    # the application side of the network
    Organizations:

    # Policies defines the set of policies at this level of the config tree
    # For Application policies, their canonical path is
    #   /Channel/Application/<PolicyName>
    Policies: &ApplicationDefaultPolicies
        Readers:
            Type: ImplicitMeta
            Rule: "ANY Readers"
        Writers:
            Type: ImplicitMeta
            Rule: "ANY Writers"
        Admins:
            Type: ImplicitMeta
            Rule: "MAJORITY Admins"
        LifecycleEndorsement:
            Type: ImplicitMeta
            Rule: "MAJORITY Endorsement"
        Endorsement:
            Type: ImplicitMeta
            Rule: "MAJORITY Endorsement"

    # Capabilities describes the application level capabilities, see the
    # dedicated Capabilities section elsewhere in this file for a full
    # description
    Capabilities:
        <<: *ApplicationCapabilities

################################################################################
#
#   SECTION: Orderer
#
#   - This section defines the values to encode into a config transaction or
#   genesis block for orderer related parameters
#
################################################################################
Orderer: &OrdererDefaults

    # Orderer Type: The orderer implementation to start
    # Available types are "solo" and "kafka"
    OrdererType: etcdraft
```

```
    Addresses:
        - orderera-pasbck-new-com:7050
        - ordererb-pasbck-new-com:7050
        - ordererc-pasbck-new-com:7050

    # Batch Timeout: The amount of time to wait before creating a batch
    BatchTimeout: 2s

    # Batch Size: Controls the number of messages batched into a block
    BatchSize:

        # Max Message Count: The maximum number of messages to permit in a batch
        MaxMessageCount: 10

        # Absolute Max Bytes: The absolute maximum number of bytes allowed for
        # the serialized messages in a batch.
        AbsoluteMaxBytes: 98 MB

        # Preferred Max Bytes: The preferred maximum number of bytes allowed for
        # the serialized messages in a batch. A message larger than the preferred
        # max bytes will result in a batch larger than preferred max bytes.
        PreferredMaxBytes: 512 KB


    # EtcdRaft defines configuration which must be set when the "etcdraft"
    # orderertype is chosen.
    EtcdRaft:
        # The set of Raft replicas for this network. For the etcd/raft-based
        # implementation, we expect every replica to also be an OSN. Therefore,
        # a subset of the host:port items enumerated in this list should be
        # replicated under the Orderer.Addresses key above.

        Consenters:
            - Host: orderera-pasbck-new-com
              Port: 7050
              ClientTLSCert: crypto-config/ordererOrganizations/pasbck-new-
com/orderers/orderera-pasbck-new-com/tls/server.crt
              ServerTLSCert: crypto-config/ordererOrganizations/pasbck-new-
com/orderers/orderera-pasbck-new-com/tls/server.crt
            - Host: ordererb-pasbck-new-com
              Port: 7050
              ClientTLSCert: crypto-config/ordererOrganizations/pasbck-new-
com/orderers/ordererb-pasbck-new-com/tls/server.crt
              ServerTLSCert: crypto-config/ordererOrganizations/pasbck-new-
com/orderers/ordererb-pasbck-new-com/tls/server.crt
            - Host: ordererc-pasbck-new-com
              Port: 7050
              ClientTLSCert: crypto-config/ordererOrganizations/pasbck-new-
com/orderers/ordererc-pasbck-new-com/tls/server.crt
              ServerTLSCert: crypto-config/ordererOrganizations/pasbck-new-
com/orderers/ordererc-pasbck-new-com/tls/server.crt
        # Options to be specified for all the etcd/raft nodes. The values here
        # are the defaults for all new channels and can be modified on a
        # per-channel basis via configuration updates.
        Options:
            # TickInterval is the time interval between two Node.Tick invocations.
            TickInterval: 500ms

            # ElectionTick is the number of Node.Tick invocations that must pass
```

```
            # between elections. That is, if a follower does not receive any
            # message from the leader of current term before ElectionTick has
            # elapsed, it will become candidate and start an election.
            # ElectionTick must be greater than HeartbeatTick.
            ElectionTick: 10

            # HeartbeatTick is the number of Node.Tick invocations that must
            # pass between heartbeats. That is, a leader sends heartbeat
            # messages to maintain its leadership every HeartbeatTick ticks.
            HeartbeatTick: 1

            # MaxInflightBlocks limits the max number of in-flight append messages
            # during optimistic replication phase.
            MaxInflightBlocks: 5

            # SnapshotIntervalSize defines number of bytes per which a snapshot is
taken
            SnapshotIntervalSize: 16 MB

    # Organizations is the list of orgs which are defined as participants on
    # the orderer side of the network
    Organizations:

    # Policies defines the set of policies at this level of the config tree
    # For Orderer policies, their canonical path is
    #   /Channel/Orderer/<PolicyName>
    Policies:
        Readers:
            Type: ImplicitMeta
            Rule: "ANY Readers"
        Writers:
            Type: ImplicitMeta
            Rule: "ANY Writers"
        Admins:
            Type: ImplicitMeta
            Rule: "MAJORITY Admins"
        # BlockValidation specifies what signatures must be included in the block
        # from the orderer for the peer to validate it.
        BlockValidation:
            Type: ImplicitMeta
            Rule: "ANY Writers"

    # Capabilities describes the orderer level capabilities, see the
    # dedicated Capabilities section elsewhere in this file for a full
    # description
    Capabilities:
        <<: *OrdererCapabilities

################################################################################
#
#   CHANNEL
#
#   This section defines the values to encode into a config transaction or
#   genesis block for channel related parameters.
#
################################################################################
Channel: &ChannelDefaults
    # Policies defines the set of policies at this level of the config tree
    # For Channel policies, their canonical path is
```

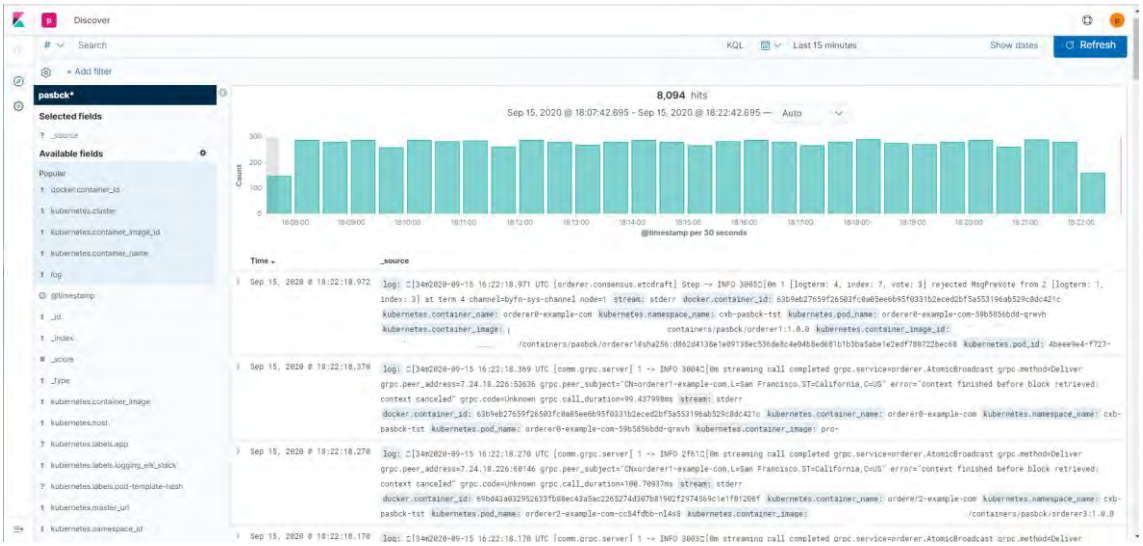
```
    #   /Channel/<PolicyName>
    Policies:
        # Who may invoke the 'Deliver' API
        Readers:
            Type: ImplicitMeta
            Rule: "ANY Readers"
        # Who may invoke the 'Broadcast' API
        Writers:
            Type: ImplicitMeta
            Rule: "ANY Writers"
        # By default, who may modify elements at this config level
        Admins:
            Type: ImplicitMeta
            Rule: "MAJORITY Admins"

    # Capabilities describes the channel level capabilities, see the
    # dedicated Capabilities section elsewhere in this file for a full
    # description
    Capabilities:
        <<: *ChannelCapabilities

################################################################################
#
#   Profile
#
#   - Different configuration profiles may be encoded here to be specified
#   as parameters to the configtxgen tool
#
################################################################################
Profiles:

    SampleMultiNodeEtcdRaft:
        <<: *ChannelDefaults
        Orderer:
            <<: *OrdererDefaults
            Organizations:
                - *OrdererOrg
            Capabilities:
                <<: *OrdererCapabilities
        Consortiums:
            SampleConsortium:
                Organizations:
                    - *Org0alfa
                    - *Org0beta
    TwoOrgsChannel:
        Consortium: SampleConsortium
        <<: *ChannelDefaults
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *Org0alfa
                - *Org0beta
            Capabilities:
                <<: *ApplicationCapabilities
```

crypto-config.yaml

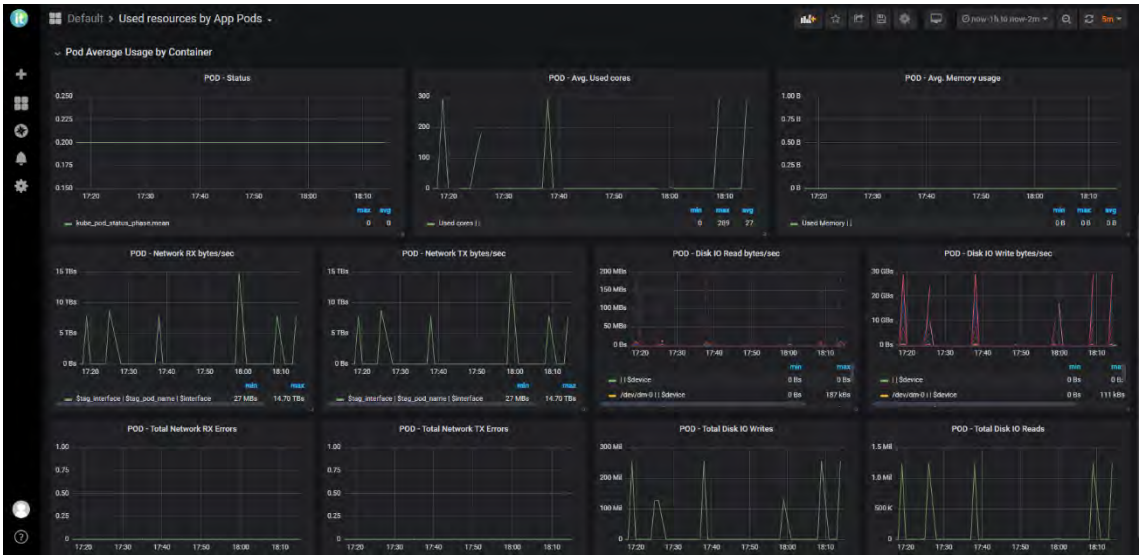
```
# ---------------------------------------------------------------------------
# ---------------------------------------------------------------------------
# "OrdererOrgs" - Definition of organizations managing orderer nodes
OrdererOrgs:
 # ---------------------------------------------------------------------------
 # Orderer
 # ---------------------------------------------------------------------------
 - Name: Orderer
   Domain: pasbck-new-com
   # ---------------------------------------------------------------------------
   # "Specs" - See PeerOrgs below for complete description
   # ---------------------------------------------------------------------------
   Specs:
     - Hostname: orderera
       CommonName: orderera-pasbck-new-com
     - Hostname: ordererb
       CommonName: ordererb-pasbck-new-com
     - Hostname: ordererc
       CommonName: ordererc-pasbck-new-com
# ---------------------------------------------------------------------------
# "PeerOrgs" - Definition of organizations managing peer nodes
# ---------------------------------------------------------------------------
PeerOrgs:
 # ---------------------------------------------------------------------------
 # Org0alfa
 # ---------------------------------------------------------------------------
 - Name: Org0alfa
   Domain: org0alfa-pasbck-new-com
   EnableNodeOUs: true
   Specs:
     - Hostname: peer0
       CommonName: peer0-org0alfa-pasbck-new-com
   Users:
     Count: 1
 # ---------------------------------------------------------------------------
 # Org0beta: See "Org0alfa" for full specification
 # ---------------------------------------------------------------------------
 - Name: Org0beta
   Domain: org0beta-pasbck-new-com
   EnableNodeOUs: true
   Specs:
     - Hostname: peer0
       CommonName: peer0-org0beta-pasbck-new-com
   Users:
     Count: 1
```

Anexo 9. Kibana y Grafana

Kibana



Grafana



Anexo 10. Build-Packs

Build.sh

```
#!/bin/sh

CHAINCODE_SOURCE_DIR="$1"
CHAINCODE_METADATA_DIR="$2"
BUILD_OUTPUT_DIR="$3"

set -euo pipefail

#external chaincodes expect connection.json file in the chaincode package
if [ ! -f "$CHAINCODE_SOURCE_DIR/connection.json" ]; then
```

```
    >&2 echo "$CHAINCODE_SOURCE_DIR/connection.json not found"
    exit 1
fi

#simply copy the endpoint information to specified output location
cp $CHAINCODE_SOURCE_DIR/connection.json $BUILD_OUTPUT_DIR/connection.json

if [ -d "$CHAINCODE_SOURCE_DIR/metadata" ]; then
    cp -a $CHAINCODE_SOURCE_DIR/metadata $BUILD_OUTPUT_DIR/metadata
fi

exit 0
```

detect.sh

```
#!/bin/sh
CHAINCODE_METADATA_DIR="$2"

set -euo pipefail

# use jq to extract the chaincode type from metadata.json and exit with
# success if the chaincode type is golang
if [ "$(cat "$CHAINCODE_METADATA_DIR/metadata.json" | sed -e 's/[{}]/''/g' | awk
-F"[,:}]" '{for(i=1;i<=NF;i++){if($i~/'type'\042/){print $(i+1)}}}' | tr -d '"')"
= "external" ]; then
    exit 0
fi
exit 1
```

release.sh

```
#!/bin/sh

set -euo pipefail

BUILD_OUTPUT_DIR="$1"
RELEASE_OUTPUT_DIR="$2"

# copy indexes from metadata/* to the output directory
# if [ -d "$BUILD_OUTPUT_DIR/metadata" ] ; then
#    cp -a "$BUILD_OUTPUT_DIR/metadata/"* "$RELEASE_OUTPUT_DIR/"
# fi

#external chaincodes expect artifacts to be placed under
"$RELEASE_OUTPUT_DIR"/chaincode/server
if [ -f $BUILD_OUTPUT_DIR/connection.json ]; then
   mkdir -p "$RELEASE_OUTPUT_DIR"/chaincode/server
   cp $BUILD_OUTPUT_DIR/connection.json "$RELEASE_OUTPUT_DIR"/chaincode/server

   #if tls_required is true, copy TLS files (using above example, the fully
qualified path for these fils would be
"$RELEASE_OUTPUT_DIR"/chaincode/server/tls)

    # copy indexes from META-INF/* to the output directory
    if [ -d "$BUILD_OUTPUT_DIR/META-INF" ] ; then
        cp -a "$BUILD_OUTPUT_DIR/META-INF/"* "$RELEASE_OUTPUT_DIR/"
    fi

   exit 0
```

```
fi
exit 1
```