



UNIVERSIDAD DE MURCIA
FACULTAD DE INFORMÁTICA

2018 - 2019

TRABAJO FIN DE GRADO

Middleware de seguridad para Blockchain en escenarios IoT

Autor
Alberto Robles Enciso

Directores
Dr. Antonio Fernando Skarmeta Gómez
Dr. Jorge Bernal Bernabé

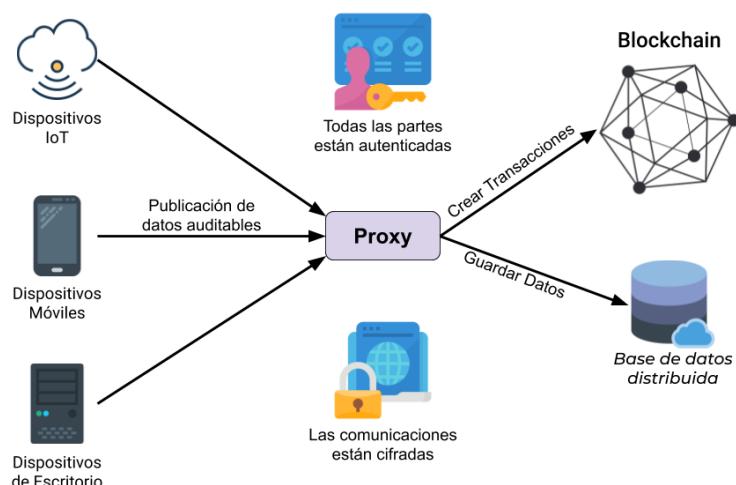
Resumen

En la actualidad estamos viviendo uno de los cambios más importantes de nuestra historia, está comenzando el inicio de la digitalización de la sociedad. Este proceso se ha acelerado gracias a la tecnología IoT (*Internet of Things - Internet de las Cosas*), su intención es que cualquier aparato disponga de pequeños ordenadores en su interior para ser capaz de acceder a Internet y comunicar los datos que producen. Esta tecnología permitirá saber en tiempo real los datos de decenas de sensores y dispositivos en instalaciones amplias (fábricas, hogares, ciudades, etc...) desde cualquier parte del mundo.

Aun así, el uso de nuevas tecnologías abre la puesta a nuevos retos y problemas, y la tecnología IoT no está exenta de ellos. Uno de los mayores retos de esta tecnología es todo lo relacionado con la seguridad y la privacidad, dado el gran número de dispositivos que integran las redes IoT, se debe gestionar de forma eficiente protocolos de seguridad que permitan proteger tanto a los dispositivos en sí como a la propia red de ataques de distinta índole, además, se debe respetar la privacidad de los datos de los usuarios que participan manteniendo sistemas de cifrado con diversas técnicas.

En el presente documento se tratarán varias propuestas para afrontar estos retos, realizando una plataforma para proporcionar un servicio de persistencia de datos seguro, auditible y escalable. El servicio deberá ofrecer un canal de comunicación encriptado y autenticado con una blockchain, y también deberá ofrecer la funcionalidad necesaria para gestionar las transacciones y realizar el proceso de autenticación.

Para realizar la solución propuesta se hará uso de la tecnología Blockchain de Hyperledger, junto con IPFS y un servicio proxy REST denominado BSH (*Blockchain Security Handler*). Gracias a estas tecnologías se creará un sistema compuesto por una blockchain para registrar transacciones, un nodo IPFS para persistir los datos de forma distribuida y un BSH que sirve de punto de acceso ligero y seguro para multitud de dispositivos clientes. En la siguiente figura se muestra el esquema simplificado del sistema que se realizará.



Esquema simplificado del escenario de uso

Además, en el documento se realizará un estado del arte que servirá para contextualizar el desarrollo de la solución explicando las tecnologías e investigaciones actuales que traten con algún tema de interés para el mismo.

Extended Abstract

We are currently experiencing one of the most important changes in our history, the beginning of digitalization. This process consists of computerizing the tasks and providing “intelligence” to the devices so that they are capable of making decisions.

With the emergence of the Internet and wireless networks, the digitalization process has accelerated and its range of action has increased, aspiring to digitize our entire society, bringing the Internet to everything around us. This had really been achieved (social networks and Internet communication) but the more industrial approach was still a little behind. After an important evolution in hardware (small-sized devices), interest in this technology began again.

At present it can be seen that this digital transformation is really happening due to the research and investment needs that arise in the national and international sphere.

This approach is called IoT (*Internet of Things*), its intention is that any device has small computers that are capable of accessing the Internet to perform data communications with a wide network of devices. This technology will allow us to know in real time the data of dozens of sensors and devices in large facilities (factories, homes, cities, etc ...) from anywhere in the world.

It could be considered that the idea of IoT will be part of the devices of the future and that all the technology will evolve to adapt to these changes, for example, along with IoT other technologies of great utility are currently appearing and that they will be a very important base for a fully connected future, among them, Bigdata is very necessary to handle the large amount of data that IoT networks can produce and Machine Learning will allow IoT networks to provide “intelligence” to be able to automatically process your data, draw conclusions from them and even take security measures if they see that there is any danger.

Even so, all innovative progress is accompanied by challenges and IoT technology is not exempt from them. One of the biggest challenges of this technology is everything related to security and privacy, given the large number of devices that integrate the IoT networks, security protocols must be managed efficiently to protect both the devices themselves and the network of attacks of different kinds, in addition, the privacy of the data of the participating users must be respected by maintaining encryption systems with various techniques.

There are currently great advances in security research in the IoT context with both traditional and innovative techniques, one of which is the use of blockchain technology. The use of systems based on the famous blockchain will allow a strong layer of security to be added to the environments where data must be shared in a distributed way, ensuring that this data will be immutable and public.

In addition, current blockchains have a number of additional tools that allow them to offer additional services such as smart contracts, private transactions and role management.

We have different motivations but, providing Internet access to devices that currently have limited resources opens up a large number of security problems, in addition, the data management itself represents a privacy problem that even now is still being studied. legal implications that may exist (GDPR).

The IoT devices usually have limited processors, which requires that they make use of reduced

languages and communication protocols and garters, on the other hand, they do not usually perform arduous encryption tasks since it requires them too much time. To simplify the tasks of the devices in the IoT networks, a “Gateway” is usually installed that acts as a receiver of data from the IoT sources and performs the most complicated tasks on that data (group them, encrypt them, send them to a database , etc...).

In the basic scheme of an IoT network, the devices communicate with the “Gateway” and make use of the credentials that they have at the factory (many manufacturers embed a certificate or a key in device) for authentication and signature operations that need, later the “Gateway” will perform the tasks of managing that data and communicating with the application to be able to offer the desired functionality. The “Gateway” is responsible for making the extra business logic required by applications developed in IoT environments.

Indeed, the basic scheme is insufficient since when we have the consolidated IoT network, new needs appear, wanting to publicly offer traceability in the data produced by the sensors, offering mechanisms to verify the integrity of the data and the origin, that the system be decentralized so as not to have a single point of failure and that the data is immutable.

Practically these needs are covered by definition Blockchain technology, thanks to its blockchain mechanism in a P2P network we can ensure that the network will be decentralized, secure, auditable, traceable and immutable.

Even so, there are still many needs to be met that will be addressed with the solution proposed in this document, among them the need to provide reliable authentication of the identity of the participants, offer auditable mechanisms for stored information (validation, integrity and provenance), offer privacy both in the data (encrypting them) and in the transaction network (permissioned blockchain) and grant acceptable performance.

To the best of our knowledge, there is currently no system that meets all these needs, there are approaches that separately meet some of them but there is no system that together offers all this functionality so the platform that will be developed will be very useful.

In addition, it should be also bear in mind that it is not recommended for reasons of efficiency to store a lot of information in blockchain transactions, so it is necessary to look for alternatives to store that data in the best possible way. An interesting solution is to use IPFS (*InterPlanetary File System*) to save the data since it follows an approach similar to blockchains, it is a decentralized, public and secure P2P data network.

If we put together all these ideas, new possibilities open up, IoT environments could be developed where all the data would be public but only the users with the appropriate keys could decrypt them, even so, any user could verify the integrity of the data and certify when they occurred, providing auditability and traceability to the information produced by customers. In addition, this system is interoperable and extensible to other modalities, instead of an IoT network, it could be considered a system of persistence of official documents, so that the blockchain could assure us immutably who (and when) created and signed a certain document giving it a legal character before any dispute.

Based on the above, we can define clear objectives that will be analyzed in the document and will be largely covered by my solution. These objectives would be:

- **Objective 1:** Look into which blockchain will be used and deploy it.

- **Objective 2:** Develop a mechanism to generate transactions in the blockchain.
 - **Objective 2.1:** Analyze what information to store in the transaction.
 - **Objective 2.2:** Investigate decentralized possibilities where to save the data that is sent.
- **Objective 3:** Research how to secure the communication and the data storage process.
 - **Objective 3.1:** Analyze authentication mechanisms of the parties involved.
 - **Objective 3.2:** Analyze data encryption mechanisms.
- **Objective 4:** Design of the solution that will perform the communication with the blockchain, client authentication, storage, signing and encryption of the data provided by the devices, generation of transactions and mechanisms to verify the data.
- **Objective 5:** Development of the Solution.
 - **Objective 5.1:** Develop an application that performs these tasks.
 - **Objective 5.2:** Develop testing software and a web client.
 - **Objective 5.3:** Test client software on machines and IoT devices.

To reach the solution proposed in this document, the work has been separated into several stages, the first of which is the longest and consists of the training and research of the technologies that are most relevant in this context. The second part is to analyze the tools provided by these technologies and make a first approach to the solution scheme. In the third stage, various solutions alternatives are designed considering what was learned in the previous stages, finally, in the fourth stage a simplified implementation of one of the schemes for a specific use case is carried out.

Everything related to the Hyperledger blockchain has been learned, in large part, in the Udemy course titled “*Blockchain Development on Hyperledger Fabric using Composer*”, which was an important training requirement because of the limited information what is on the Internet and the difficulty of understanding that Hyperledger concepts initially have. The course explains how Hyperledger Fabric works, how it is deployed and how applications are created using Hyperledger Composer.

In the development of the solution a spiral cycle has been followed, so that the development has been separated into four repeated phases in an incremental way, in this way a series of objectives must first be defined, after that the possible solutions are analyzed, implemented and finally tested. When the last phase is completed correctly it starts again adding more difficult objectives.

Initially the development has focused on creating the basic skeleton of the application and in subsequent cycles functionality has been added to the methods. In this way, the priority in each cycle has been to independently carry out the most important methods (authentication, data transmission, etc ...) to leave each functionality completely complete before starting another.

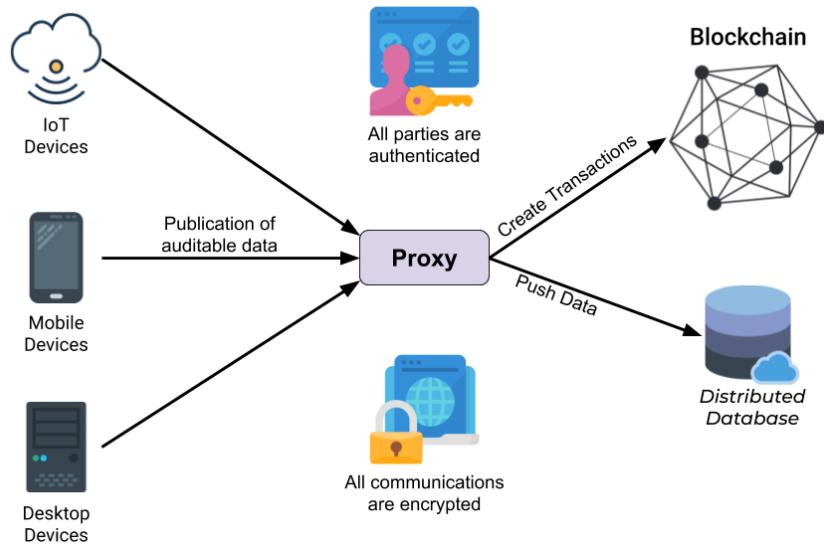
In the development of this document we will study the type of blockchain that will be used in the proposed solution, this will be a private blockchain. Specifically, the blockchain to be used will be Hyperledger Fabric, from the Hyperledger team, along with the Hyperledger Composer development tools.

A blockchain is a data structure in which the information is separated into blocks which contain data and also metadata to refer to the temporarily previous block, this means that with simple cryptography techniques any modification on a block requires editing all the blocks after this. This property, being computationally very expensive, allows blockchain technology to be used as a distributed and public database that will store a history of immutable data.

In practice this technology was used to develop (with asymmetric cryptography and hash functions) a distributed “accounting record” (*ledger*) that guaranteed that the immutability of the data without requiring any trusted entity. This requires developing a P2P network to distribute the blockchain and perform consensus tasks among the members of the network, this is very important because of the security implications it has.

The blockchain technology is interesting in environments where it is required to store data in a continuous and public way over time, without the possibility of modification and whose trust is intended to be distributed instead of residing in a certifying entity.

Regarding the development of the platform, it should be taken into account that the proposed objectives and the various specifications must be achieved, the following figure shows a very simplified use scenario. The purpose of the platform is to allow IoT devices to be freed from the tasks of interacting directly with the blockchain, decoupling functionality in the proxy.



Simplified scheme of the use scenario

In summary, a series of requirements arise from the objectives set and based on the needs of the system:

- **Requirement 1:** Transactions persisted on the platform must be confidential, there should be no clear record of the author.
- **Requirement 2:** Customers who participate must be authenticated, clearly showing to the platform information to identify themselves. Authentication can be delegated to external systems following a federated approach.
- **Requirement 3:** The proxy must be designed to support the use by IoT clients with limited capabilities (constrained devices).

- **Requirement 4:** The confidentiality of persisted data must be ensured, for this it is necessary that they be stored encrypted.
- **Requirement 5:** The data that persists must be validated to ensure its integrity.
- **Requirement 6:** Metadata should be recorded in transactions that provide information on the origin, for traceability reasons, of the persisted data.
- **Requirement 7:** The platform must be efficient, for this it must be fast and with light transactions (little data and persistence *offchain*).
- **Requirement 8:** The privacy of the participants and their transactions must be ensured, so there is no record of information that allows the authors of the transactions to be easily identified. The meta-data of the sixth requirement must be adequate to respect this privacy requirement without losing traceability.

Following the proposals for solutions that meet the objectives and requirements, two application designs have been devised, both of which share a similar structure since their main difference is in the authentication process. The solution is called “Blockchain Security Handler” and two alternatives are made, one that will manage the authentication process and another that delegates it to another external system.

After all the above, a series of conclusions can be drawn and objectives can be defined to perform in the future.

It can be concluded that all the stated objectives have been achieved. Thanks to the research carried out, much has been learned about Hyperledger, about security issues in IoT environments and the implementation of a verifiable example platform has been carried out in a practical use case. At the same time, a state of the art of current technologies has been realized to contextualize the design of the solution.

As a future work we have everything that is left out of study or realization, the most immediate is the implementation of the federated design for the realization of authentication using eIDAS together with FiWare. Another issue that was initially discussed was to perform the authentication process through smart contracts, but this was not finally done by preferring the current scheme, we could study how to implement this alternative in the current solution.

To conclude, the document is structured as follows: **Section 2** (State of the Art) a state of the art will be realized where the technologies related to the realization of this project will be seen in order to provide a base for the reader to allow it understanding of it. Also, this section includes the final part a state of the art in which the background is detailed within the framework of public research. Then, in the **section 3** (Analysis and Design of the Solution) the design alternatives will be analyzed and the architecture proposed for the solution will be explained, the tools and technologies needed to perform the design will also be briefly explained solution. In the **section 4** (use-case) the software development of the design proposed in the previous section is described in detail, and its deployment and validation for a specific use case as a test together with some practical examples. And finally, in the **section 5** (conclusion and future work) the final conclusions will be shown, as well as the suggested future paths for this project. Finally, a series of *Annexes* are provided, with detailed information on the Hyperledger Fabric deployment, the installation of a BNA (*Business Network Application*) and the deployment of the proxy service.

Siglas

Listado de las siglas y acrónimos que se usarán en las explicaciones de este documento:

- **API:** Application Programming Interface.
- **ARM:** Advanced RISC Machine.
- **BNA:** Business Network Application.
- **BSH:** Blockchain Security Handler.
- **CA:** Certification Authority.
- **CID:** Content Identifier.
- **CP-ABE:** Ciphertext-Policy Attribute-Based Encryption.
- **DApps:** Decentralized Applications.
- **DID:** Decentralized Identifiers.
- **DLT:** Distributed Ledger Technology.
- **DNI:** Documento Nacional de Identidad.
- **DNIE:** Documento Nacional de Identidad electrónico.
- **eIDAS:** electronic IDentification, Authentication and trust Services.
- **FNMT:** Fábrica Nacional de Moneda y Timbre.
- **HTTP:** Hypertext Transfer Protocol.
- **HTTPS:** Hypertext Transfer Protocol Secure.
- **IdM:** Identity management.
- **ILP:** Interledger Protocol.
- **IoT:** Internet of Things.
- **IPFS:** InterPlanetary File System.
- **JS:** JavaScript.
- **JSON:** JavaScript Object Notation.
- **JWT:** JSON Web Token.
- **MSP:** Membership Service Provider.
- **P2P:** Peer-to-Peer.
- **PEM:** Privacy-Enhanced Mail.
- **PoC:** Proof of Concept.
- **PoET:** Proof of Elapsed Time.
- **PoW:** Proof Of Work.
- **REST:** Representational state transfer.

- **SAML:** Security Assertion Markup Language.
- **SBC:** Single-Board Computer.
- **SCADA:** Supervisory Control And Data Acquisition.
- **SOAP:** Simple Object Access Protocol.
- **SSH:** Secure Shell.
- **SSL:** Secure Sockets Layer.
- **TFG:** Trabajo fin de Grado.
- **TLS:** Transport Layer Security.
- **VM:** Virtual Machine.
- **XML:** Extensible Markup Language.

Índice de contenidos

1	Introducción	1
1.1	Motivación	2
1.2	Objetivos	3
1.3	Metodología	3
1.4	Estructura del documento	4
2	Estado del Arte	5
2.1	Introducción a la tecnología Blockchain	5
2.1.1	Hyperledger	7
2.2	IPFS	15
2.3	Autenticación Federada	15
2.4	Otras tecnologías	16
2.4.1	API REST	16
2.4.2	Java EE	16
2.4.3	Certificados X.509	17
2.4.4	Cifrado simétrico y asimétrico	17
2.4.5	Vagrant	17
2.5	Estado del arte científico	18
3	Análisis y diseño de la solución	19
3.1	Análisis de necesidades	20
3.2	Propuesta de solución	22
3.3	Diseño final	26
3.3.1	Diseño del esquema centralizado	28
3.3.2	Diseño del esquema federado	30
3.4	Diseño de clientes	32
3.4.1	Dispositivo Personal	32
3.4.2	Dispositivo IoT	34
4	Implementación del caso de uso	36
4.1	Blockchain	36
4.2	Blockchain Security Handler	38
4.2.1	Modelo	39
4.2.2	Utilidades	40
4.2.3	Servicio REST	44
4.3	Implementación de los Clientes	48
4.3.1	Ejemplo Java	48
4.3.2	Ejemplo Web	48
4.3.3	Ejemplo IoT	50
4.4	Resumen final de la implementación realizada	52
5	Conclusiones y Vías futuras	54
	Referencias	56
	Anexos	58
	Anexo I: Despliegue de Hyperledger	58

Anexo II: Creación de una Business Network Application	62
Anexo III: Iniciar Composer REST con autenticación JWT	64
Anexo IV: Iniciar nodo IPFS	67
Anexo V: Iniciar Servicio Java y probarlo	69
Anexo VI: Instalar la aplicación cliente en una Raspberry	72

Índice de figuras

1	Esquema de proyectos dentro de Hyperledger	7
2	Hyperledger Greenhouse	9
3	Inicio de una transacción desde un cliente o una aplicación	13
4	Flujo del protocolo de Hyperledger Fabric, obtenido de la cita [1]	14
5	Esquema simplificado del escenario de uso	20
6	Arquitectura de la interacción de Keyrock con eIDAS, fuente [2]	23
7	Flujo de comunicación de Keyrock con eIDAS, fuente [2]	23
8	Descripción de la estructura del CID	24
9	Esquema base del diseño final	26
10	Esquema simplificado de los componentes del <i>Blockchain Security Handler</i>	27
11	Esquema del diseño centralizado	29
12	Flujo de comunicaciones del diseño centralizado	29
13	Esquema del diseño federado	31
14	Flujo de comunicaciones del diseño federado	31
15	Ejemplo de escenario de persistencia de documentos oficiales	33
16	Raspberry Pi Zero W	34
17	Ejemplo de escenario IoT	35
18	Esquema de despliegue virtualizado de Hyperledger Fabric	36
19	Modelo SQL de la base de datos	41
20	Ejemplo de aplicación Web	49
21	Variable de Entorno en Windows	60
22	Asistente Yeoman para crear una BNA de Hyperledger Composer	62
23	Estructura del directorio de una BNA creada con Yeoman	62
24	Resultado de compilar e instalar una BNA	63
25	Resultado de iniciar una BNA e importar la Card de administrador	63
26	Página jwt.io	65
27	Página jwt.io con el ejemplo del proxy	66
28	Directorio de Go-IPFS	67
29	Resultado de la ejecución del nodo IPFS	67
30	Panel de administración del nodo IPFS	68

1 Introducción

La necesidad de automatizar tareas existe desde el inicio de las tecnologías, en su origen las máquinas de vapor ayudaron a la industria y posteriormente a la sociedad a evolucionar, los motores de combustión permitieron otro cambio importante, dando paso también al vehículo personal. Tras la invención del transistor y la creación de los ordenadores de propósito general se consiguió uno de los cambios más importantes de nuestra historia, comenzó el inicio del proceso de digitalización.

Este proceso consistió inicialmente en informatizar tareas simples, después se logró abarcar tareas más complejas y finalmente se consiguió dotar de “inteligencia” a los dispositivos para que sean capaces de tomar decisiones ellos mismos.

Con la aparición de Internet y las redes inalámbricas el proceso de digitalización se aceleró y aumentó su rango de acción, aspirando a digitalizar toda nuestra sociedad llevando Internet a todo lo que nos rodea. Esta aspiración se consigue alcanzar (dispositivos móviles, redes sociales y comunicación por Internet) pero el enfoque más industrial se había quedado un poco atrás. Tras conseguir una evolución importante en el hardware (dispositivos de tamaño reducido) comenzó de nuevo el interés por esta tecnología en el ámbito empresarial.

En la actualidad se puede apreciar que realmente está ocurriendo esta transformación digital por las necesidades de investigación e inversión que surgen en el ámbito nacional[3] e internacional.

Este enfoque se denomina IoT (*Internet of Things - Internet de las Cosas*), su intención es que cualquier aparato disponga de pequeños ordenadores que sean capaces de acceder a Internet para realizar comunicaciones de datos con una amplia red de dispositivos[4]. Esta tecnología nos permitirá saber en tiempo real los datos de decenas de sensores y dispositivos en instalaciones amplias (fábricas, hogares, ciudades, etc...) desde cualquier parte del mundo.

Se podría considerar que la idea de IoT formará parte de los dispositivos del futuro y que toda la tecnología evolucionará para adaptarse a estos cambios, por ejemplo, a la par de IoT están apareciendo otras tecnologías de gran utilidad actualmente y que serán una base muy importante para un futuro totalmente conectado, entre ellas se puede destacar Bigdata por ser muy necesaria para manejar la gran cantidad de datos que pueden producir las redes IoT[5] y por otro lado tenemos el Machine Learning que permitirá dotar a las redes IoT de “inteligencia” para poder procesar de forma automática sus datos[6], sacar conclusiones de ellos[7] e incluso tomar medidas de seguridad si ven que existe algún peligro[8].

Aun así, todo avance innovador va acompañado de retos y la tecnología IoT no está exenta de ellos. Uno de los mayores retos de esta tecnología es todo lo relacionado con la seguridad y la privacidad, dado el gran número de dispositivos que integran las redes IoT se debe gestionar de forma eficiente protocolos de seguridad que permitan proteger tanto a los dispositivos en sí como a la propia red de ataques de distinta índole, además, se debe respetar la privacidad de los datos de los usuarios que participan manteniendo sistemas de cifrado con diversas técnicas.

Actualmente hay grandes avances en investigación sobre seguridad en el contexto IoT tanto con técnicas tradicionales como con técnicas innovadoras, una de ellas es la del uso de la tecnología blockchain. El uso de sistemas basados en la famosa cadena de bloques permitirá añadir una fuerte capa de seguridad a los entornos donde se deban compartir unos datos de forma distribuida, asegurando que esos datos serán inmutables y públicos.

Además, las blockchains actuales poseen una serie de herramientas adicionales que les permiten ofrecer servicios extras como contratos inteligentes, transacciones privadas y gestión de roles.

1.1 Motivación

Dotar de acceso a Internet a dispositivos que, en la actualidad, disponen de recursos limitados abre una gran cantidad de problemas de seguridad, además, el manejo en sí de los datos representa un problema de privacidad que incluso ahora sigue en estudio las implicaciones legales que pueden existir (RGPD).

Los dispositivos IoT suelen tener procesadores limitados, lo que requiere que hagan uso de lenguajes y protocolos de comunicación reducidos y ligeros, por otro lado, no suelen realizar tareas de encriptación arduas ya que les requiere demasiado tiempo. Para simplificar las tareas de los dispositivos en las redes IoT se suele instalar un “Gateway” que hace de receptor de datos de las fuentes IoT y realiza las tareas más complicadas sobre esos datos (agruparlos, encriptarlos, enviarlos a una base de datos, etc...).

En el esquema básico de una red IoT, los dispositivos se comunican con el “Gateway” y hacen uso de las credenciales que disponen de fábrica (muchos fabricantes incrustan un certificado o una clave en dispositivo) para las operaciones de autenticación y firma que necesiten, posteriormente el “Gateway” realizará las tareas de gestión de esos datos y las de comunicación con la aplicación para poder ofrecer la funcionalidad deseada. El “Gateway” se encarga de hacer la lógica de negocio extra que requieran las aplicaciones desarrolladas en los entornos IoT.

Aun así, el esquema básico es insuficiente ya que cuando tenemos la red IoT consolidada aparecen nuevas necesidades, querer ofrecer de forma pública trazabilidad en los datos que producen los sensores, ofrecer mecanismos para verificar la integridad de los datos y la procedencia, que el sistema sea descentralizado para no tener un único punto de fallo y que los datos sean inmutables.

Prácticamente estas necesidades las cubren por definición la tecnología Blockchain, gracias a su mecanismo de cadena de bloques en una red P2P podemos asegurar que la red será descentralizada, segura, auditible, traceable e inmutable.

Aun con lo dicho, quedan muchas necesidades pendientes de cubrir que se intentaran abordar con la solución propuesta en este documento, entre ellas se puede destacar la necesidad de proporcionar una autenticación fiable de la identidad de los participantes, ofrecer mecanismos de auditabilidad de la información almacenada (validación, integridad y procedencia), ofrecer privacidad tanto en los datos (cifrándolos) como en la red de transacciones (blockchain con privilegios) y otorgar un rendimiento aceptable.

No existe en la actualidad un sistema que satisfaga todas esas necesidades, existen aproximaciones que por separado cumplen algunas de ellas, pero no existe ningún sistema que en conjunto ofrezca toda esta funcionalidad por lo que la plataforma que se desarrollará será de gran utilidad.

También se tiene que tener en cuenta que no es recomendable, por motivos de eficiencia, almacenar mucha información en las transacciones de la blockchain, por lo que es necesario buscar alternativas para almacenar esos datos de la mejor manera posible. Una solución interesante es usar IPFS (*InterPlanetary File System - Sistema de archivos interplanetarios*) para guardar los datos ya que sigue un enfoque semejante a las blockchains, es una red de datos P2P descentralizada, pública y segura[9].

Si juntamos todas estas ideas se abren nuevas posibilidades, se podrían desarrollar entornos IoT donde todos los datos serían públicos pero solo los usuarios con las claves adecuadas podrían desencriptarlos, aun así, cualquier usuario podría verificar la integridad de los datos y certificar cuando se produjeron, proporcionando auditabilidad y trazabilidad a la información que producen los clientes. Además, este sistema es interoperable y extensible a otras modalidades, en vez de una red IoT, se podría considerar un sistema de persistencia de documentos oficiales, de tal forma que la blockchain nos pudiese asegurar de forma inmutable quién (y cuándo) creó y firmó un determinado documento dándole un carácter legal ante cualquier disputa.

1.2 Objetivos

Por lo comentado anteriormente podemos definir unos objetivos claros que serán analizados en el documento y serán cubiertos en gran medida por mi solución. Dichos objetivos serían:

- **Objetivo 1:** Estudiar que blockchain se usará y desplegarla.
- **Objetivo 2:** Desarrollar un mecanismo para generar transacciones en la blockchain.
 - **Objetivo 2.1:** Estudiar qué información almacenar en la transacción.
 - **Objetivo 2.2:** Estudiar posibilidades descentralizadas donde guardar los datos que se envían.
- **Objetivo 3:** Estudiar como segurizar todo el proceso de comunicación y almacenamiento de datos.
 - **Objetivo 3.1:** Estudiar mecanismos de autenticación de las partes involucradas.
 - **Objetivo 3.2:** Estudiar mecanismos de encriptación de los datos.
- **Objetivo 4:** Diseño de la solución que realizará las tareas de comunicación con la blockchain, autenticación de los clientes, almacenamiento, firmado y encriptación de los datos que proporcionan los dispositivos, generación de transacciones y mecanismos para verificar los datos.
- **Objetivo 5:** Desarrollo de la solución.
 - **Objetivo 5.1:** Desarrollar una aplicación que realice estas tareas.
 - **Objetivo 5.2:** Desarrollar software de pruebas y un cliente web.
 - **Objetivo 5.3:** Pruebas de los clientes en equipos y dispositivos IoT.

1.3 Metodología

Para llegar a la solución propuesta en este documento se ha separado en varias etapas el trabajo, la primera de ellas es la más larga y consiste en la formación e investigación de las tecnologías que son más relevantes en este contexto. La segunda parte consiste en analizar las herramientas que proporcionan esas tecnologías y hacer un primer planteamiento del esquema de la solución. En la tercera etapa se diseñan diversas alternativas de soluciones teniendo en cuenta lo aprendido en las etapas anteriores, por último, en la cuarta etapa se realiza una implementación simplificada de uno de los esquemas para un caso de uso concreto.

Todo lo relacionado con la blockchain de Hyperledger se ha aprendido, en gran parte, en el curso de Udemy titulado “*Blockchain Development on Hyperledger Fabric using Composer*”[10], el cual fue un requisito de formación importante por la escasa información que hay en Internet y la dificultad de comprensión que tienen inicialmente los conceptos de Hyperledger. En el curso se explica cómo funciona Hyperledger Fabric, como se despliega y como se crean aplicaciones usando Hyperledger Composer.

En el desarrollo de la solución se ha seguido un ciclo en espiral, de forma que se ha separado el desarrollo en cuatro fases repetidas de forma incremental, de esta manera primero se deben definir una serie de objetivos, tras eso se analizan las posibles soluciones, después se implementan y finalmente se prueban. Cuando se termina la última fase correctamente se vuelve a empezar añadiendo objetivos de mayor dificultad.

Inicialmente el desarrollo se ha centrado en crear el esqueleto básico de la aplicación y en los ciclos posteriores se le han ido añadiendo funcionalidad a los métodos. De esta forma la prioridad en cada ciclo ha sido la de realizar de forma independiente los métodos más importantes (autenticación, envío de datos, etc...) para dejar cada funcionalidad totalmente completa antes de comenzar con otra.

1.4 Estructura del documento

El presente documento se ha estructura en cuatro secciones en las cuales se va analizando y modelando la solución desarrollada.

Así pues, en la **sección 2** se realizará un estado del arte donde se verán las tecnologías relacionadas con la realización de este proyecto con el fin de proporcionar una base al lector que le permita una correcta comprensión del mismo. Asimismo, esta sección incluye la parte final un estado del arte científico en cual se detallan los antecedentes dentro del marco de investigaciones públicas.

En la **sección 3** se analizaran las alternativas de diseños y se explicará la arquitectura propuesta para la solución, también se explicaran brevemente las herramientas y tecnologías necesarias para realizar la solución.

En la **sección 4** se describe en detalle el desarrollo software del diseño propuesto en la sección anterior, y su despliegue y validación para un caso de uso concreto a modo de prueba junto con algunos ejemplos prácticos.

Y, por último, en la **sección 5** se mostrarán las conclusiones finales, así como las vías futuras sugeridas para este proyecto.

Para terminar, también se ofrecen una serie de *Anexos*, con información detallada del despliegue de Hyperledger Fabric, la instalación de una BNA (*Business Network Application*) y el despliegue del servicio proxy.

2 Estado del Arte

En esta sección se va a introducir las tecnologías utilizadas para el análisis y desarrollo de la solución propuesta, esto servirá de base para tener una correcta comprensión de las tecnologías y la terminología que usaremos en las siguientes secciones. Además, al final de esta sección se incluye un estado del arte científico dónde se citan diversos artículos de investigación que tratan temas relacionados con la solución propuesta.

2.1 Introducción a la tecnología Blockchain

Una cadena de bloques (*blockchain*) es una estructura de datos en la cual se separa la información en bloques los cuales contienen datos y además metadatos para hacer referencia al bloque temporalmente anterior, esto logra que con técnicas simples de criptografía cualquier modificación sobre un bloque requiera editar todos los bloques posteriores a este. Esta propiedad, al ser computacionalmente muy costosa, permite que se pueda usar la tecnología blockchain como una base de datos distribuida y pública que almacenará un histórico de datos inmutables.

En la práctica esta tecnología se usó para desarrollar (gracias a la criptografía asimétrica y las funciones hash) un “registro contable” (*ledger*) distribuido que garantizase que la inmutabilidad de los datos sin necesidad de requerir ninguna entidad de confianza. Esto requiere desarrollar una red P2P para distribuir la cadena de bloques y realizar las tareas de consenso entre los miembros de la red, esto es muy importante por las implicaciones de seguridad que tiene[11].

La tecnología blockchain es interesante en entornos en los que se requiera almacenar de forma pública y continuada datos ordenados en el tiempo, sin posibilidad de modificación y cuya confianza pretenda ser distribuida en lugar de residir en una entidad certificadora.

Por estos requisitos surgen varios aspectos a tener en cuenta en esta tecnología:

- **Almacenamientos de datos:** Se consigue forzando a cada nodo de la red a contener una copia completa de la cadena de bloques actual.
- **Transmisión de datos:** Se logra con el desarrollo de una red de pares (P2P) donde cada nodo puede comunicarse con cualquiera de los demás y entre ellos distribuyen la información.
- **Confirmación de datos:** Para lograr esto se requiere un mecanismo de consenso entre los nodos participantes. El mecanismo más utilizado en las blockchain más populares es POW (*Proof Of Work - Prueba de Trabajo*) en el que se les requiere a los nodos realizar un trabajo para demostrar su interés legítimo en participar, de esta forma validan las entradas que se añaden en los nuevos bloques en un proceso conocido como Minería o Minar.

La primera aplicación práctica de esta tecnología y actualmente la más popular fue en el desarrollo de la criptodivisa Bitcoin en 2009 [12].

En Bitcoin los datos que se almacenaban se denominaron transacciones ya que representaban traspasos de la divisa homónima y los bloques representan conjuntos de transacciones que deben ser firmadas (minado). En otras implementaciones (como Hyperledger Fabric) se extiende la información que se puede almacenar, además de las transacciones se permite guardar contratos inteligentes y “entidades” que se van modificando haciendo uso de los mismos o de transacciones,

de tal forma que la blockchain permite modelar una base de datos de entidades con un histórico de su evolución detallado, público e inmutable.

Blockchain públicos y privados

Se pueden separar las blockchains en dos categorías según los requisitos de acceso a su red P2P, si son de acceso libre se denominan públicas y si no privadas.

Las **blockchains públicas** fueron las primeras en aparecer (p.e., Bitcoin y Ethereum) y se basan en que el acceso a la red sea totalmente abierto, permitiendo que cualquiera pueda participar en la red con el rol que quiera. En estas redes es muy importante el número de usuarios para mantener un correcto funcionamiento, para ello motiva la participación de los nodos que realizan las verificaciones y el consenso (mineros) a través de incentivos (pagos en la criptodivisa). Esto es lo que ocurre en Bitcoin, donde los usuarios que participan en la red (mineros) son recompensados con bitcoins cuando realizan correctamente el cálculo de la firma de un bloque.

El problema de estas redes es que el número de nodos se masifica, si bien esto es una ventaja en el ámbito de la seguridad de la red, en el caso de la velocidad y el coste de la red es un problema ya que se debe mantener limitada las recompensas por el minado de tal forma que cada vez se requiere un coste computacional mayor para añadir cada bloque y la velocidad de las transacciones comienza a disminuir (y en el caso de Bitcoin aumentan también las comisiones de la transacción).

Otro problema que tienen, por definición, es que son públicas por lo que no existe ningún tipo de privacidad en las transacciones, no existen criterios para decidir quien participa en la red y todos los nodos son anónimos. Por ello se considera que este tipo de blockchain son interesantes para proyectos de dominio público, pero en el ámbito empresarial no son interesantes.

Por otro lado tenemos las **blockchains privadas**, normalmente éstas están desarrolladas por y para las empresas, de tal forma que el acceso a la red requiere de unas credenciales que la propia red valida asegurando así la privacidad de la información almacenada en la blockchain. A este tipo de red blockchain se le suele llamar red autorizada/permisionada (*Permissioned*) ya que pone restricciones a quién puede unirse, lo que pueden ver y lo que pueden hacer, logrando una capa de privacidad extra. En estas redes los nodos siguen teniendo la tarea del mantenimiento de la cadena de bloques, pero al estar autenticados todos ellos (no son anónimos) pueden aparecer roles especiales entre ellos, algunos se pueden encargar de las verificaciones de autenticación, de la creación de credenciales, de la creación de bloques, del mecanismo de consenso, del proceso de commit de los bloques y cualquier operación restringida que se requiera.

Esta modalidad de red permite que las transacciones sean muy rápidas por tener un número reducido de nodos y algoritmo de consenso más ligero (posiblemente sin una ardua prueba de trabajo). En el mundo empresarial estas redes tienen un gran interés y por ello actualmente los principales desarrolladores y usuarios son grandes empresas que aspiran a que la tecnología blockchain sea un estándar en las industrias y revolucionen nuestro día a día[13].

Existen varios ejemplos de blockchains privadas, incluso algunas desarrolladas sobre blockchains públicas, pero de ellas las más populares en el ámbito empresarial son las desarrolladas por el equipo Hyperledger.

Este trabajo se centrará en las blockchains *Permissioned* por el enfoque industrial que se le dará.

2.1.1 Hyperledger

Hyperledger es un proyecto de código abierto creado por la Fundación Linux en diciembre de 2015. Nació con el objetivo de crear un ecosistema centrado en DLTs (*Distributed Ledger Technology - Tecnología distribuida del libro mayor, otra forma de llamar a las blockchain privadas*) opensource de ámbito corporativo.

Su línea de trabajo principal fue la de combinar todos los esfuerzos de los socios para conseguir desarrollar protocolos y estándares abiertos en el ámbito de la tecnología de las blockchain privadas.

El proyecto Hyperledger agrupa una gran variedad de negocios basados en la tecnología blockchain, incluyendo frameworks, librerías, interfaces gráficas, redes blockchain privadas, motores de Smart Contracts o aplicaciones. Su objetivo es el de seguir una estrategia modular siendo posible la reutilización de módulos comunes y el intercambio de unos módulos por otros que se adecuen más a nuestras necesidades.

En sus inicios, el proyecto Hyperledger fue lanzado por 30 miembros fundadores que colaboraron en la creación de Hyperledger Fabric, por parte Digital Asset Holding, libconsensus de Blockstream y OpenBlockchain de IBM, y por otro lado Hyperledger Sawtooth liderado por Intel.

Dentro del proyecto, en la actualidad, hay un gran número de miembros en constante crecimiento, teniendo entre ellos empresas muy influyentes dentro del mundo de las tecnologías como Cisco, IBM, Intel y SAP o dentro del mundo financiero como J. P. Morgan, SWIFT y BBVA

Dentro de Hyperledger tenemos 14 proyectos distintos separados en dos categorías, frameworks blockchain y herramientas[14]. En la **figura 1**, versión ampliada de la fuente [15], podemos ver un esquema simplificado de la separación de los distintos proyectos.

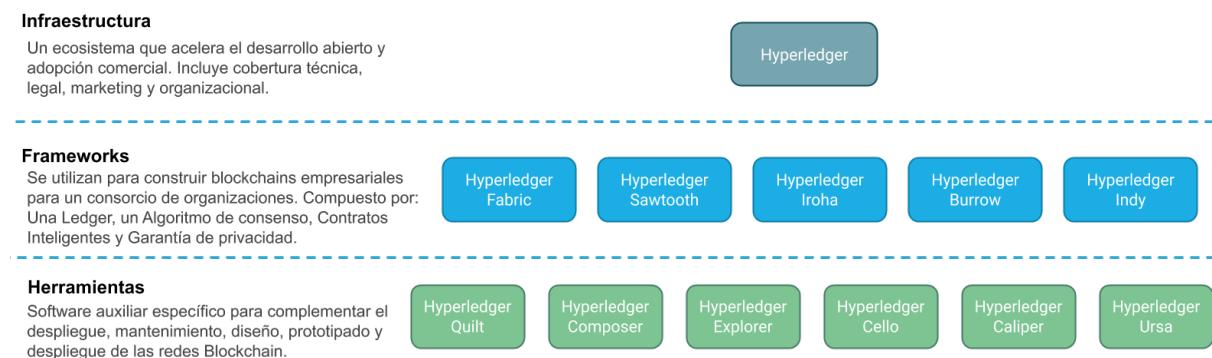


Figura 1: Esquema de proyectos dentro de Hyperledger

Los frameworks son diferentes implementaciones de la tecnología blockchain, cada uno con un enfoque totalmente distinto [16][17]. Estos son:

- **Hyperledger Fabric:** El proyecto más popular dentro de Hyperledger, tanto que normalmente, de forma errónea, se denomina a esta blockchain simplemente Hyperledger. Es una blockchain de carácter privado orientada al mundo empresarial y fue desarrollada inicialmente por IBM. Es una plataforma multidisciplinaria que permite el despliegue de smart contracts (chaincodes) desarrollados en el lenguaje de programación de Google (Golang), aunque actualmente permite otros lenguajes como NodeJS, Java y JavaScript. Actualmente es la blockchain privada más conocida.

- **Hyperledger Sawtooth:** Proyecto principalmente desarrollado por Intel. Es también una blockchain modular privada de ámbito empresarial altamente configurable que incorpora la capacidad de despliegue de smart contracts en varios lenguajes y permite cambiar de forma dinámica el algoritmo de consenso, el más común es PoET (“*Proof of Elapsed Time*”).
- **Hyperledger Iroha:** Pretende ser una plataforma blockchain simple y modular para ser fácilmente incorporada en otros proyectos, que permita el despliegue de Smart Contracts desarrollados en Java, muy enfocada a las aplicaciones móviles. Esta plataforma está desarrollada por Soramitsu, Hitachi, NTT Data y Colu.
- **Hyperledger Burrow:** Conocida anteriormente como Monax, fue impulsada por la startup Monax Industries y es una blockchain privada basada en el código de Ethereum. Es decir, permite el despliegue de Smart Contracts desarrollados en Solidity.
- **Hyperledger Indy:** En una plataforma cuyo principal objetivo es proponer una solución de Identidad Digital distribuida (DID)[18]. Proporciona herramientas, librerías y componentes reutilizables para proveer identidades digitales arraigadas en la cadena de bloques o en otros ledgers distribuidos de modo que sean interoperables en distintos dominios administrativos y aplicaciones. Apuesta por un protocolo de conocimiento cero (“*Zero Knowledge Protocol*”), este protocolo tiene como objetivo dar la menor cantidad de información posibles en procesos en los que usa la identidad digital.

Por otro lado, tenemos los proyectos que consisten en herramientas que sirven de apoyo para el desarrollo y uso los frameworks, estas herramientas se desarrollan de forma separada maximizando el enfoque modular de Hyperledger. Hay un total de 8 herramientas, a continuación se explican 6 de ellas:

- **Hyperledger Quilt:** Es una herramienta que ofrece interoperabilidad entre sistemas DLTs que utiliza el protocolo ILP (Interledger Protocol). Su objetivo es conseguir una comunicación efectiva entre distintas redes blockchain. El protocolo ILP fue creado por Ripple para realizar una transacción entre dos ledger distintas de redes distintas (y con implementaciones distintas).
- **Hyperledger Composer:** Es una herramienta que permite la creación de aplicaciones descentralizadas (denominada en Hyperledger como *Business Network Application*, BNA), abstrandose del desarrollo de bajo nivel. Permite programarlas el chaincode (Smart Contract) en varios lenguajes, aunque habitualmente se hace uso de JavaScript. Para modelar las entidades dispone de un lenguaje simplificado propio.
- **Hyperledger Explorer:** Aplicación web que proporciona una interfaz visual para la monitorización de los nodos de Hyperledger Fabric, bloques, estadísticas, transacciones, contactos inteligentes y mucho más. Su intención es servir de plataforma de pruebas y depuración.
- **Hyperledger Cello:** Herramienta que intenta reducir el esfuerzo de la creación y administración de una blockchain y sus servicios para intentar lograr un modelo de despliegue BassS (“*Blockchain as a Service*”)
- **Hyperledger Caliper:** Herramienta de benchmarking para las plataformas blockchain. Analiza el grado de rendimiento de las blockchains en función de un conjunto de casos de uso predefinidos. Produce informes con indicadores de rendimiento como transacciones por segundo, la latencia de la transacción, etc.... Su objetivo es producir una serie de datos

que sirvan para analizar que blockchain se ajusta mejor a nuestras necesidades.

- **Hyperledger Ursa:** Es una librería criptográfica compartida, modular y flexible.

Ahora que ya conocemos que es Hyperledger y que proyectos forman parte de él podemos tratar los dos que hemos usado para la realización de este TFG. En la **figura 2** podemos apreciar el conjunto completo de proyectos de Hyperledger, lo que ellos denominan “*Hyperledger Greenhouse*”.

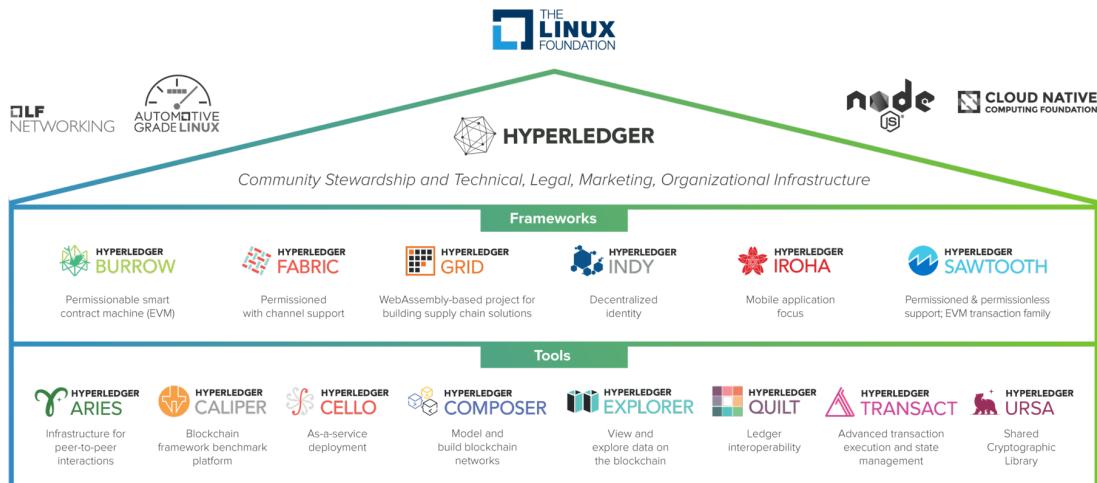


Figura 2: Hyperledger Greenhouse

Hyperledger Composer

Hyperledger Composer es un proyecto de Hyperledger que tiene como finalidad ofrecer herramientas para desarrollar aplicaciones abstrayéndose de la parte de programación nativa de Hyperledger Fabric. Gracias a Composer cuando queramos desarrollar una aplicación para Fabric nos centraremos únicamente en la lógica de diseño de activos, participantes y transacciones.

Gracias a esta característica el desarrollo de DApps (“*distributed applications*”, en Hyperledger denominadas BNA) se hace de una manera mucho más rápida, en comparación con el proceso de desarrollo de las aplicaciones tradicionales. Además, Composer ofrece una API de alto nivel que facilita enormemente el despliegue y mantenimiento de la aplicación en la red Fabric.

El hecho de usar un lenguaje propio simplificado para modelar las entidades y lenguajes populares para la implementación del chaincode nos da los siguientes beneficios:

- **Facilita la compresión de la tecnología de las BNAs:** Permite hacer fácilmente prototipos reducidos para probar la funcionalidad y no requiere de altos conocimientos de programación para implementarlos y desplegarlos.
- **Disminución del tiempo de desarrollo:** El uso de la herramienta permite crear aplicaciones rápidamente, mediante un asistente y pocas líneas de código. Este tiempo es significativamente inferior al que se necesitaría para hacer un desarrollo nativo en Hyperledger Fabric ya que tendría que definir una arquitectura de red, sus controladores y sus Smart Contracts (chaincodes) desarrollados en el lenguaje Golang, lenguaje que no es comúnmente conocido.

- **Reduce el riesgo:** A la hora de realizar el proyecto muchos de sus componentes son reusables y ha sido pensado para que fuera fácil de depurar y probar.
- **Flexible:** Gracias a la capacidad de abstracción y simplificación que genera esta herramienta la solución desarrollada o parte de ella es fácilmente adaptable para ser integrada en otras aplicaciones.

Por lo dicho se puede considerar Composer como la herramienta fundamental para trabajar con las blockchains de Hyperledger. Composer además de ofrecer un marco de desarrollo cómodo también proporciona un servicio REST que ofrece una API sencilla de usar para las aplicaciones de escritorio que desarrollemos nosotros.

Hyperledger Fabric

Como ya se ha comentado, Hyperledger Fabric es uno de los proyectos más importantes de Hyperledger y consisten en un DLT con fines empresariales, su enfoque es el de desarrollar una blockchain modular y escalable enfocada en los negocios, de tal forma que esta será privada y “*permissioned*”, restringiendo el acceso de quien puede participar y autenticando a los participantes para saber quién es cada uno y los permisos que tiene (*access control*). También es programable por lo que permite la creación de Smart Contracts (*Chaincodes*) y su ejecución al realizar una transacción.

La blockchain Fabric permite, gracias a su mecanismo de autenticación (MSP) poder identificar a los nodos de la red lo permitirá añadir reglas de seguridad para hacer transacciones privadas, para poder realizar contratos confidenciales, o incluso poner reglas de visibilidad creando canales (dentro de la red) privados entre varias entidades. A su vez, el esquema seguido no requiere de ninguna criptodivisa ya que no necesita ningún incentivo para premiar a los nodos que realizan la validación, esto ocurre porque no se sigue un esquema de consenso basado en una prueba de trabajo (POW), por el contrario, sigue un esquema de validación más simple y ligero ya existen una serie de nodos que se encargan en exclusiva de realizar el consenso y otros la validación, más adelante se explicarán en detalle.

En los siguientes apartados se explicará cómo funciona en detalle Hyperledger Fabric, pero lo primero que es necesario conocer son los conceptos básicos, si bien anteriormente se han usado o se han comentado en parte, ahora se les debe dar el enfoque de la blockchain Fabric.

Conceptos de Hyperledger Fabric

El objetivo de cualquier aplicación es cumplir un propósito, en el caso de la blockchain se busca modelar datos, en Fabric a esos datos se les denomina **Assets**, estos representan un *valor* en el mundo físico que se desea guardar en la blockchain con su información, por ejemplo, un coche con su matrícula (identificador) y su dueño (estado) o una casa. Por lo dicho, la ejecución de transacciones realiza cambios en los Assets, por lo que podemos considerar las transacciones como el mecanismo para cambiar el estado de los Assets, por ejemplo, al vender el coche a una persona realizamos una transacción para indicar el cambio del dueño del coche (no se crea un nuevo Asset, es el mismo, pero se cambia el valor del dueño).

Junto a los Assets se tiene el **Chaincode**, este tiene varias finalidades, por una lado define la estructura de los Assets (los modela) e indica qué transacciones puede tener, por otro lado, define la lógica de negocio que se ejecutará en forma de Smart Contracts. Al realizar la transacción ejecutamos un cambio en los Assets el cual se modela con la ejecución de un chaincode concreto.

Dentro de chaincodes se pueden implementar también reglas de privacidad o permisos, roles y peticiones de búsqueda de datos personalizadas. El resultado de compilar un chaincode es una BNA.

Como ya se dijo, Fabric es un DLT por lo que todos los nodos de la red disponen de una copia del Ledger o libro de transacciones, en este se tiene un listado completo de las transacciones que se han hecho (en forma de blockchain) de forma para determinar el estado de los Assets se tiene que recorrer toda la lista hasta el final.

La utilidad de esto es que se tiene un histórico exacto e inmutable de los estados que ha tenido cada Asset, muy necesario para temas de *provenance*, como por ejemplo, en la fabricación de un vehículo (representa un Asset) se podría representar las distintas partes que tiene (con otro Asset) y se podría saber exactamente de qué fábricas procede cada una desde su fabricación inicial hasta el usuario final.

Dentro de una red Fabric existen diferentes tipos de nodos que realizan tareas distintas, esto es posible ya que la identidad de todos los nodos es conocida porque, como se indicó al principio, es una red “*permissioned*”. La identificación de los nodos se logra al dotarles de identidad por medio de certificados **X.509**, el servicio que se encarga de esto es el **MSP** (“*Membership Service Provider*”).

El MSP es un componente que define las reglas para validar, autenticar y permitir el acceso a la red a una identidad o participante otorgándoles roles y reglas de acceso. El MSP usa una Autoridad de certificación (CA) y el interfaz por defecto es Fabric-CA API. Este componente es fácilmente reemplazable lo que hace a Hyperledger Fabric muy flexible a la hora de usar un mecanismo de identificación u otro. Esta CA crea los certificados como una CA normal, pudiendo renovarlos y revocarlos. Es posible que en una misma red existan varios MSPs.

Los miembros son las entidades que participan en la blockchain, estas se consideran como entidades legalmente separadas (empresas o usuarios distintos), a cada uno de ellos se les otorga un certificado X.509 validado y generado por el MSP para que realicen con él las tareas de autenticación y firma. Además, en Fabric también los nodos de la red tienen un certificado para asegurar su identidad.

Los participantes se conectan a la red haciendo uso de un nodo en concreto junto con su certificado propio de usuario, el cual usan para firmar la transacción que generan. El certificado del nodo lo usa la red para saber si se debe confiar en el nodo (es válido), en caso de que el nodo no tenga un certificado valido la transacción no se acepta, aunque el certificado del usuario sea válido.

Tras la correcta identificación de los nodos de la red Fabric surgen diferentes roles entre ellos, estos son necesarios por la arquitectura blockchain que impone Hyperledger Fabric.

Los nodos son considerados como los End-Point de la blockchain, los cuales usan un protocolo P2P para comunicarse y distribuir el Ledger, en el caso, por ejemplo, de Bitcoin los nodos se comunican el Ledger y realizan las validaciones (minado), en Hyperledger Fabric se sigue un esquema muy distinto, estos son una entidad de comunicación, que mantiene la red operativa y tiene una copia del Ledger.

A continuación, se explicarán los distintos tipos de nodos de los que dispone Hyperledger Fabric.

Roles en Hyperledger Fabric

En la red Fabric disponemos de nodos que realizan el rol de **Client** (*Cliente*), estos son constituyen la parte más externa de la red y sirve de punto de comunicación con las aplicaciones externas. Su finalidad es la de ofrecerse como punto de inicio de las transacciones de los usuarios (o aplicaciones), recibiendo la transacción firmada por el usuario y tras verificarla la firman y la envían a la red para comprobar su validez. Es decir, estos nodos permiten a los usuarios finales la comunicación con la blockchain de forma sencilla.

Otro de los roles que pueden tener los nodos es **Peer** (*Par*), estos serán los nodos internos de la red, encargándose de realizar el protocolo P2P para distribuir y mantener actualizado el Ledger entre ellos. Básicamente estos nodos representaran el concepto descentralizado de la blockchain, por ello interesa tener el máximo número posible para poder distribuir el trabajo y evitar un único punto de fallo.

Dentro de los nodos con el rol de *Peer* existen tres tipos especiales. Por un lado tenemos los **Endorsers Peers**, estos se encargan de recibir inicialmente la transacción desde un *Client* y su tarea es validarla, para ello comprueban los certificados y las firmas que se hayan aplicado sobre la transacción y simulan la ejecución del chaincode para ver si funciona correctamente pero no guardan el estado (no cambian nada). Su finalidad es proteger a la red siendo una primera barrera de confirmación.

Por otro lado tenemos los **Committers Peers** que se encargan de hacer la parte final de una transacción, estos reciben la petición de grabado de un bloque en la blockchain por parte del *Ordering Service* por lo que validad si el resultado de la transacción es correcto y proceden a guardar el bloque y distribuirlo por la red.

Además, existe otro tipo de *Peer* más simple que está relacionado con este, el **Anchor Peer**, los nodos de este tipo representan un nodo frontera en las redes de una organización (dentro de una red global de Fabric) de tal manera que son los nodos a los que los *Committers Peers* envían el bloque para que estos lo distribuyan en su red.

Paralelo a estos nodos tenemos otros que realizan un rol de gran importancia, estos son los **Orderers Peers** que juntos forma el denominado **Ordering Service**. Este servicio es la *backbone* de la red, es el responsable de asegurar la consistencia del Ledger y mantener el orden de las transacciones, los nodos que lo forman reciben las transacciones validadas de los *Endorsers* y las ordenan para formar un bloque, realizan el mecanismo de consenso y aseguran la atomicidad de las transacciones. Tras crear el bloque y ejecutar los *chaincodes* asociados a las transacciones se envía el bloque resultante a los *Committers* para que lo persistan.

Los *Orderers Peers* se comunican con un protocolo escalable de paso de mensajes, la implementación por defecto de Fabric es **SOLO** de tal forma que solo existe un único nodo *Orderer* lo que simplifica el funcionamiento y las pruebas, pero si queremos un escenario más realista varios *Orderers* para que sea un proceso distribuido y escalable debemos usar **Apache Kafka** como *Ordering Service*.

Así pues, sabiendo los distintos roles que toman los nodos de una red de Hyperledger Fabric, se puede explicar el proceso de funcionamiento que tienen usando de ejemplo el flujo de comunicaciones que ocurre al realizar una transacción desde una aplicación externa.

Funcionamiento de Hyperledger Fabric

Cuando un cliente (o una aplicación) desea realizar una transacción lo primero que debe hacer es crear la transacción y firmarla con su certificado, esto lo puede hacer fácilmente gracias a Hyperledger Composer, además Composer permite enviar esa transacción a un nodo *Client* de la red. En la **figura 3** se muestra esta interacción.

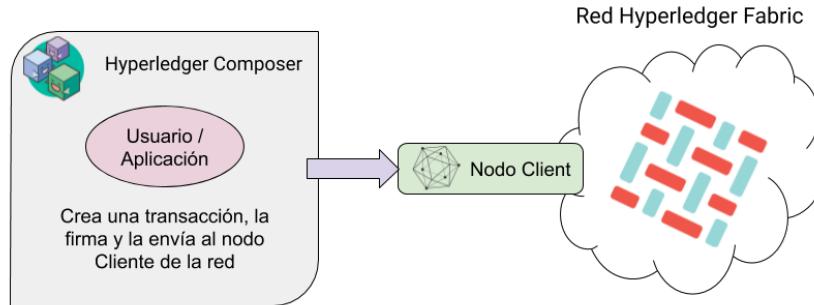


Figura 3: Inicio de una transacción desde un cliente o una aplicación

Al recibir el nodo *Client* la transacción, este procede a validar la autenticación del usuario y la validez de la firma de la transacción, si todo es correcto se considera que la transacción puede entrar a la red y comienza el flujo de funcionamiento de Fabric, en caso de que falle la validación se notifica directamente al usuario.

Cuando se considera la transacción válida para ser propuesta a la red comienza el siguiente protocolo (en la **figura 4** se muestra gráficamente):

1. **Propuesta de transacciones:** El cliente procede a crear una transacción con la información y la firma del usuario, firma con su certificado esa transacción y envía una propuesta de transacción a los *Orderers Peers*.
2. **Simulación y respaldo de transacciones:** Cada uno de los *Orderers Peers* que recibe esa propuesta de transacción procede a simularlas teniendo en cuenta el estado actual del Ledger pero sin realizar ningún cambio sobre el mismo, tras eso genera un paquete denominado RW Set que contiene lista de *Reads and Writes* (lecturas y escrituras) generados por la transacción simulada. El RW Set es firmado por el Endorser y devuelto al cliente.
3. **Recepción de aprobaciones:** El cliente espera el resultado de los *Orderers Peers* para saber si aprueban su transacción.
4. **Publicación de la transacción:** El cliente, al tener el visto bueno de los *Orderers Peers* procede a enviar transacción firmada por el Endorser y el RW Set al Ordering Service, el cual es común para toda la red, para guardar la transacción en el Ledger.
5. **Ordenación de transacciones:** El *Ordering Service* al recibir una transacción la guarda en un bloque ordenando las que tengan en orden de creación, después envía el bloque a todos los *Committers* (y estos los envían a los *Anchor Peers* de cada organización). Hyperledger Fabric incluye a día de hoy tres mecanismos de ordenación para implementar este servicio: SOLO, Kafka y SBFT (Simplified Byzantine Fault Tolerance). Como ya dijimos SOLO es útil para pruebas, y para un escenario real deberíamos usar Kafka.
6. **Verificar bloques:** Los *Committers* comprueban entonces que los RW Sets recibidos aún son válidos y generan la misma lista de *Reads and Writes*. Sin una transacción resulta

invalida durante este proceso será incluida en el bloque, pero marcada como inválida y no modifica el estado del registro.

7. **Notificación:** Por último, los Committers informan al Cliente de si la transacción ha sido ejecutada con éxito o no y este transmitirá el resultado al usuario o aplicación que le pidió crear la transacción.

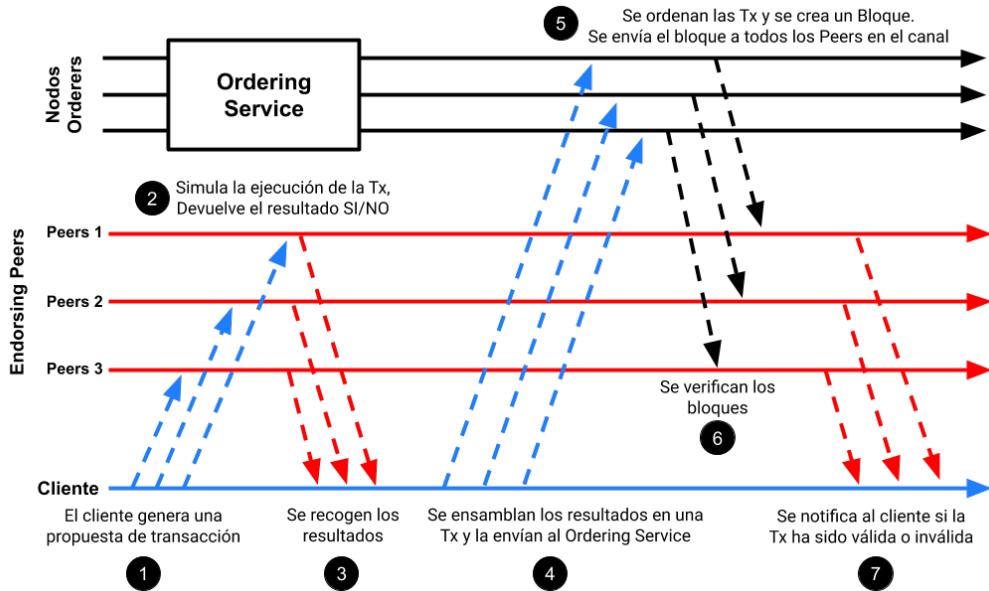


Figura 4: Flujo del protocolo de Hyperledger Fabric, obtenido de la cita [1]

Canales en Hyperledger Fabric

Como se ha dicho, Fabric está pensada para el ámbito empresarial por lo que debe ofrecer herramientas para añadir más privacidad a las comunicaciones, esto lo consigue implementando un mecanismo de canales privados entre entidades permitiendo tener diferentes blockchains en la misma red de forma que solo los participantes de un canal pueden conocer los detalles de las transacciones que ocurren en dicho canal.

Así pues, un miembro al conectarse a la red debe seleccionar el canal donde trabajará (por defecto existe un único canal disponible para todos), las transacciones que se hagan en ese canal estarán aisladas del resto de canales. Por cada canal se mantiene un Ledger independiente y entre ellos no hay ningún tipo de visibilidad. Es posible que un miembro esté en varios canales a la vez, pero siempre respetando los Ledge de cada uno y manteniendo las transacciones de cada canal separadas.

Si dos entidades que están dentro de la red quisiesen hacer un intercambio (realizar una transacción) pero que tanto la información transmitida como el intercambio sean privados (solo ellos dos lo sepan) deberán crear un canal privado entre ellos y realizar la transacción ahí. De esa forma ellos dos tendrían constancia de la transacción ya que tienen acceso a ese canal pero el resto de miembro de la red no recibirían nada y tampoco podrían unirse a ese canal al no tener privilegios en el mismo, haciendo que esa transacción sea secreta.

2.2 IPFS

Con la creciente evolución de las redes de datos descentralizadas y con el ejemplo de redes P2P altamente funcionales (como *BitTorrent*) apareció un interés en conseguir desarrollar una red P2P de archivos distribuido direccional por contenido para almacenar y compartir hipermedia con el objetivo de sustituir la web (*HTTP*) por una web descentralizada.

De esta idea nació **IPFS** (*InterPlanetary File System - Sistema de archivos interplanetarios*), un sistema de archivos distribuidos punto a punto que busca conectar todos los dispositivos en un mismo sistema de archivos proporcionando un sistema de almacenamiento en bloques de la información, de tal forma que cada nodo cliente deberá guardar la información que considere interesante pidiéndola a sus vecinos. Este enfoque es semejante al de una red Torrent en la que todos los clientes formasen parte del mismo *swarm* y compartiesen los ficheros entre ellos.

En protocolo IPFS identifica los ficheros con un hash (comúnmente se utiliza como **CID** un *multipath*) lo que permite fácilmente referenciar a los datos y evitar de forma sencilla los duplicados en la red. Es interesante el uso de IPFS como una base de datos “gratuita” donde con gran facilidad podemos guardar datos de forma distribuidos, pública e inmutable. Las características de IPFS son muy semejante a lo que aporta una blockchain pero sin la limitación en el volumen de datos a almacenar, por eso en la práctica suelen ser tecnologías que van juntas cuando se requiere persistencia de mucha información en las transacciones haciendo que dentro de ellas vaya, en vez de todos los datos, una referencia (CID) al datos en IPFS.

2.3 Autenticación Federada

Por lo general, la autenticación y autorización de usuarios en los sistemas convencionales se realiza a través de un base de datos específica o un servicio de directorio activo mantenido por el administrados del sistema, lo que representa un punto de seguridad crítico y es poco escalable a sistemas extensibles.

Si quisiésemos unificar la autenticación de diversos sistemas en un sistema interoperable encargado exclusivamente de esta tarea, siendo extensible a otras plataformas y aplicaciones deberíamos usar una **Autenticación Federada**.

La autenticación federada consiste en delegar a una entidad de confianza las tareas de obtener y guardar información de usuarios para su autenticación. La aplicación sigue manteniendo el control de autorizar qué recursos puede ver el usuario, pero se libera de la necesidad de autenticar cada usuario y asignarle los correspondientes roles o grupos convirtiéndose en un sistema de autenticación pasiva. Esto permite desacoplar la autenticación de la autorización y evitar que cada aplicación tenga que gestionar las credenciales de sus usuarios.

El mecanismo típico de funcionamiento de estos sistemas es el uso de **tokens** que representan una “clave” para autenticar en nuestro sistema proporcionada por el sistema de autenticación federada.

Lo interesante es que el mecanismo de autenticación federada puede ser mantenido por una gran comunidad y apoyado por las administraciones de los países de tal forma que se ofrezca un sistema de autenticación delegado con carácter legal.

Existe un ejemplo de estos por parte de la plataforma **FIWARE**, esta es una plataforma, impulsada por la Unión Europea, para el desarrollo y despliegue global de aplicaciones de IoT.

FIWARE intenta proveer de una arquitectura totalmente abierta, pública y libre, así como de un conjunto de especificaciones que permita a los desarrolladores, proveedores de servicios, empresas y otras organizaciones desarrollar productos en el marco del IoT. Dentro de las herramientas de FIWARE está el componente **KeyRock**, el cual es un IdM (*Identity Management*) que sirve de módulo para realizar las operaciones de gestión de credenciales y cumplir el papel de Autenticación Federada dentro de nuestro entorno IoT.

Lo interesante de KeyRock es que, a su vez, dispone de un conector que permite delegar la autenticación al sistema **eIDAS** (“*electronic IDentification, Authentication and trust Services*” - “*Servicio electrónico de Identificación, Autenticación y confianza*”), esto permite que nuestro sistema federado delegue en uno de mayor rango de acción, logrando que sea posible autenticarse en nuestros sistemas haciendo uso de credenciales europeas (**DNIe**).

Esta tecnología, en conjunto, ofrece unas posibilidades muy interesantes ya que nos permite ofrecer servicios en los que se legitime las acciones de los usuarios dentro del marco europeo con gran facilidad. Por ejemplo, sería posible hacer uso del certificado incluido en el DNIe para el proceso de autenticación y la clave privada para todos los procesos de firma de documentos en sistemas que requieran persistir datos.

2.4 Otras tecnologías

A continuación mostramos otras tecnologías que aparecerán a lo largo del trabajo y de las cuales puede resultar útil conocerlas brevemente.

2.4.1 API REST

Una API (*Application Programming Interface - Interfaz de Programación de Aplicaciones*) es un conjunto de especificaciones y métodos que soporta una biblioteca que sirve de “interfaz” o “conector” entre el código de la biblioteca y el nuestro.

Una API REST es un tipo de API que se basa en que la comunicación se haga sobre HTTP, haciendo uso de los mensajes que ofrece el protocolo (GET, POST, PUT, UPDATE y DELETE) para dar semántica a las operaciones, las cuales hacen uso de la URL para diferenciarse de forma verbose. El resultado típico de REST suelen ser documentos XML o JSON.

En la actualidad son muy utilizadas este tipo de API por la ventaja de ser muy simples de usar en prácticamente cualquier dispositivo y cualquier lenguaje ya que reutilizan toda la tecnología web ya existente.

2.4.2 Java EE

Java Enterprise Edition es una versión de la popular plataforma de programación Java enfocada a las aplicaciones empresariales, ofrece herramientas y librerías para realizar aplicaciones distribuidas y modulares. Uno de los servicios más interesante de Java EE es el soporte para aplicaciones Web, permitiendo programar y desplegar servicios web en diversos estándares (JSP, JSF, Java-RS), en nuestro caso será de especial utilidad el soporte de Servlet para desplegar un servicio REST siguiendo el estándar Java-RS (Jersey) en el contenedor Apache Tomcat.

2.4.3 Certificados X.509

Los certificados X.509 son un tipo de certificado que sigue el estándar UIT-T para infraestructuras de claves públicas, sirven para asociar una clave pública a una identidad, guardando en el certificado todos los datos relativos a la misma.

Los certificados X.509 se generan a través de una entidad de confianza conocida como autoridad de certificación (CA). La CA administra uno o varios certificados especiales llamados certificados de CA, que utiliza para generar certificados X.509. Solo la autoridad de certificación tiene acceso a los certificados de CA. Las CAs representan entidades de confianza común para poder demostrar que un certificado es lícito, firmando con su clave privada los certificados que expiden.

2.4.4 Cifrado simétrico y asimétrico

Dentro del campo de la criptografía, el cifrado es el procedimiento para transformar un mensaje en otro que sea incompresible o, al menos, difícil de comprender, haciendo uso de un determinado algoritmo de cifrado con una cierta clave. Para lograr el proceso contrario es necesario conocer la clave secreta del algoritmo, esta puede ser la misma que se usó para cifrar los datos (criptografía simétrica) u otra distinta (criptografía asimétrica).

Las dos técnicas de cifrado típicas son la simétrica y la asimétrica. El cifrado simétrico es el más simple, se hace uso de una única clave para todo el proceso que deberán conocer todas las partes implicadas (secreto compartido), es rápido y fácil de usar pero su seguridad depende por completo de la distribución de la clave. Las técnicas más famosas son AES, DES, 3DES o RC4.

El cifrado asimétrico requiere el uso de dos claves, una privada que solo conoce el dueño de la clave y otra pública que cualquiera puede tener, el proceso de cifrado se basa en propiedades matemáticas que permite que al combinar ambas claves se consiga el mensaje original permitiendo que se pueda cifrar con cualquiera de las dos claves pero que sea necesaria la otra para descifrar. La técnica más famosa es RSA.

2.4.5 Vagrant

Vagrant es una herramienta, desarrollada por HashiCorp, que nos ayuda a crear y gestionar máquinas virtuales desde un terminal. Nos permite definir los servicios a instalar, así como también sus configuraciones desde un único fichero, facilitando enormemente el despliegue de máquinas virtuales con configuraciones específicas. Está pensado para trabajar en entornos locales y lo podemos utilizar con shell scripts o la línea de comandos.

La gran ventaja de Vagrant es que posee un archivo de configuración *Vagrantfile* donde se centraliza toda la configuración de la VM que creamos, de forma que se puede utilizar el fichero *vagrantfile* para crear una VM exactamente igual todas las veces que sea necesario y en los equipos que queramos, dando un factor de portabilidad al entorno de trabajo muy interesante.

Cabe destacar que Vagrant no tiene la capacidad para ejecutar una máquina virtual, sino que simplemente se encarga de las características con las que debe crearse esa VM y los complementos a instalar. Para poder trabajar con las máquinas virtuales es necesario instalar un hipervisor como VirtualBox.

2.5 Estado del arte científico

Creemos que actualmente no existe una solución que cubra todas las necesidades de esta línea de investigación. No obstante existen diversos artículos donde se tratan por separado algunas de las soluciones que propongo o analizan ideas semejantes que me han ayudado a lograr mi solución.

En el artículo “*Blockchain based Proxy Re-Encryption Scheme for Secure IoT Data Sharing*”[19] se introducen los problemas de seguridad que tiene IoT y proponen usar una blockchain para dar más seguridad, la idea es hacer uso de una aplicación intermedia (denominada “proxy”) que realice tareas de encriptación y almacenamiento de datos en una Cloud y utilice el mecanismo de contratos inteligentes de la blockchain para gestionar el proceso de obtención de claves públicas. De este artículo he sacado la idea de una aplicación intermedia que ofrezca la funcionalidad extra y enlace la red IoT con la blockchain, también la idea de realizar un proceso extra de encriptación o firma de los datos y el envío de los mismo a una Cloud (como IPFS).

También en el artículo “*Cloud Data Provenance using IPFS and Blockchain Technology*”[20] se analiza la forma más eficiente de persistir datos de una blockchain en IPFS, como se comentó, es muy típico usar ambas tecnologías juntas en los escenarios donde tengamos que persistir datos. En el artículo “*IoT data privacy via blockchains and IPFS*”[9] se puede ver el esquema más común que se suele usar para hacerlo.

Respecto el apartado de autenticación, todavía es un tema muy nuevo y aún más aplicado los temas IoT junto con Keyrock. Tenemos artículos como “*The Use of the Blockchain Technology for Public Key Infrastructure of eIDAS*”[21] donde se muestran las primeras ideas de hacer uso de eIDAS dentro de la tecnología blockchain, por otro lado, en el artículo “*Authenticated Academic Services through eIDAS*”[22] se muestra el uso típico del sistema eIDAS para la autenticación de un microservicio.

Sobre la gestión de la privacidad en IoT utilizando plataformas como Keyrock hay mucha menos investigación, un artículo interesante, realizado por investigadores de nuestra facultad, es “*Holistic Privacy-Preserving Identity Management System for the Internet of Things*”[23] donde se usa Keyrock junto con Idemix.

Actualmente existe bastante investigación en temas de seguridad para entornos IoT, pero esta sigue enfoques tradicionales por lo al aplicar nuevas tecnologías, como el caso de la blockchain, surgen retos aún sin resolver, en el artículo “*IoT Security: Review, Blockchain Solutions, and Open Challenges*”[24] se tratan las soluciones actuales que existen y los problemas que surgen de su uso.

Por otro lado, en “*A survey on the security of blockchain systems*”[25] se introduce a la tecnología blockchain, mostrando su evolución desde una criptodivisa hasta un nuevo enfoque en los sistemas distribuidos. En el artículo se muestran las alternativas de blockchains populares y las implicaciones en seguridad que se deben tener en cuenta en cada una.

Como ya se dijo, se aprecia como este tema aún es muy reciente por lo que la investigación es escasa y no cubre todos los temas que nos interesa, aun así, de los artículos por separado se pueden sacar ideas para juntarlas en un único sistema que agrupe las innovaciones de cada *topic* dando como resultado una plataforma innovadora que cumpla las especificaciones del sistema a desarrollar.

3 Análisis y diseño de la solución

Una vez explicadas las tecnologías implicadas en este proyecto, podemos pasar al análisis y diseño de la solución que propongo. Al finalizar esta sección se tendrá un diseño de solución claro que permitirá satisfacer el **objetivo 4** y dará paso a la realización de una implementación.

Uno de los primeros aspectos que se deben analizar es la elección de blockchain para el proyecto, dentro de los candidatos tenemos el referente **Bitcoin**, por otro lado **Ethereum** y finalmente **Hyperledger Fabric**.

	Acceso	Moneda	Transacciones	Comisiones	Smart Contracts	Privacidad
Bitcoin	Pública	Si	Muy lentas	Muy Altas	No	No
Ethereum	Pública	Si	Lentas	Altas	Si	No
Fabric	Privada	No	Muy Rápidas	Ninguna	Si	Si

La opción de Bitcoin es interesante por la gran red que tiene y la amplia comunidad de usuarios, pero realmente no es muy útil ya que es una red blockchain bastante primitiva (no ofrece contratos inteligentes), por otro lado, por motivo de la especulación monetaria es una red muy saturada con transacciones lentas y costosas comisiones. Con lo dicho nos damos cuenta que Bitcoin no es un candidato válido ya que tecnológicamente no cumple nuestras especificaciones.

Otro candidato bastante interesante es Ethereum, esta blockchain es mucho más sofisticada ya que permite contratos inteligentes en un lenguaje bastante potente, por otro lado, dispone de una amplia comunidad y una gran cantidad de nodos pero sigue dependiendo de una criptomoneda, está algo menos saturada que Bitcoin pero también es un foco de especulación y tiene unas comisiones de minado no despreciables.

El último candidato es Hyperledge Fabric, el cual sabemos de antemano que es la mejor opción, que destaca por ser de base una blockchain con enfoque empresarial y sin ningún tipo de criptodivisa lo que evita la especulación y la necesidad de comisiones para minar. Además, como se dijo en la sección anterior, permite la creación de contratos inteligentes de alta funcionalidad y soporta medidas de privacidad y seguridad muy útiles para las aplicaciones empresariales.

De esto podemos concluir que la blockchain que se usará será Hyperledge Fabric, cumpliendo así el **objetivo 1**, junto con las herramientas de Hyperledger Composer para facilitar el desarrollo y despliegue de la BNA. En los anexos se adjunta el procedimiento para montar la blockchain, para desplegar una BNA y para iniciar el servicio REST de Composer.

En Hyperledger Fabric disponemos de dos maneras de realizar el despliegue, una manera es en un entorno nativo con un equipo con el propósito exclusivo de albergar la blockchain y la otra manera es en una máquina virtual preinstalada y configurada en cualquier equipo. Será necesario, en función de nuestros requisitos, elegir la manera más adecuada.

	Rendimiento	Escalabilidad	Portabilidad	Despliegue
Nativo	Muy Alto	Alta	Baja	Difícil
Virtual	Alto	Media	Muy Alta	Fácil

La nativa consiste en instalar en un equipo todo el software que requiere Hyperledger de forma manual y configurar todos los requisitos de Fabric y Composer. La alternativa virtual permite

desplegar el servicio dentro de una máquina virtual (Virtualbox) creada y mantenida por Vagrant, de forma que las configuraciones necesarias son mínimas ya que el resto está automatizado.

La opción escogida ha sido la virtual, aunque la opción nativa ofrece un rendimiento notoriamente superior y está pensada para entornos en producción para nosotros no es muy relevante ya que tenemos unas necesidades reducidas, nos limitaremos a hacer uso de la funcionalidad básica para un caso de uso pequeño por lo que la opción virtual es útil ya que nos permite realizar pruebas y reinstalarlo todo muy rápidamente, además permite que el entorno sea portable y realiza la configuración de los requisitos de máquina virtual de forma automática.

El lenguaje de programación que se usará para desarrollar la aplicación *proxy* y uno de los clientes será Java ya que es un lenguaje potente, popular y multiplataforma. Para la implementación del servicio *proxy* se hará uso de la librería Jersey que es una implementación del estándar Java-RS. El cliente web estará programado en HTML y JavaScript ya que son los lenguajes típicos en el desarrollo web.

3.1 Análisis de necesidades

El sistema a desarrollar deberá cumplir varias especificaciones y objetivos, en la **figura 5** se muestra un escenario de uso muy simplificado. La finalidad de la plataforma es permitir liberar a los dispositivos IoT de las tareas de interactuar directamente con la blockchain, desacoplando la funcionalidad en el proxy.

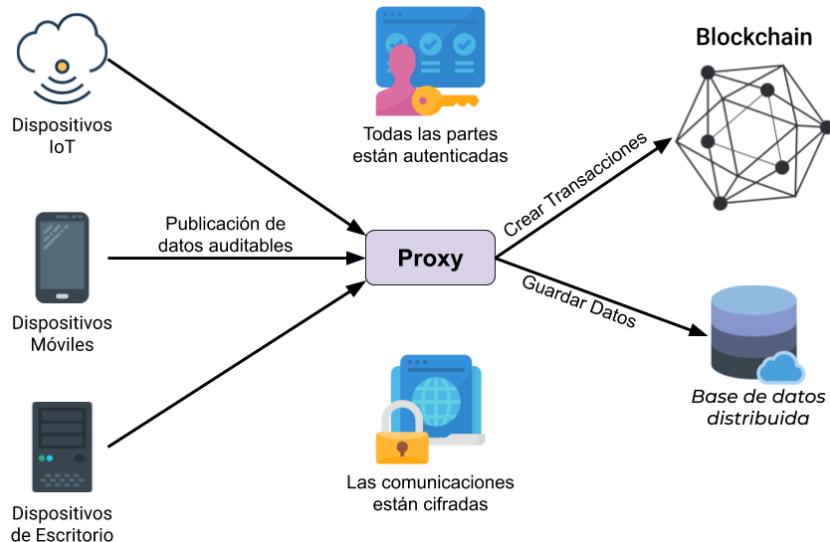


Figura 5: Esquema simplificado del escenario de uso

La primera necesidad importante que surge en base al **objetivo 1** es el despliegue de una blockchain para registrar las operaciones que haga el sistema, por ello será necesario desarrollar una comunicación entre el sistema y la blockchain, requiriendo autenticar y cifrar el canal de comunicación para que sea seguro y solo nuestro sistema pueda usarlo.

Paralelamente se necesita un soporte IPFS para guardar los datos cifrados de forma distribuida para lograr el **objetivo 2.2**.

En el desarrollo de la plataforma se debe tener en cuenta que su uso preferente será desde dispositivos IoT o clientes básicos, por lo que el sistema deberá ofrecer un acceso simple y ligero. Además, el acceso deberá ser seguro, de forma que el canal de comunicación deberá estar protegido y el cliente tendrá que autenticarse.

Por la necesidad autenticación y cifrado se debe realizar un diseño que proponga directamente los mecanismos necesarios para conseguir autenticar a los clientes participantes para cumplir el **objetivo 3**.

Se deberá estudiar que operaciones ofrecerá el sistema y que datos se guardarán en la blockchain para cubrir las especificaciones. Es necesario que la aplicación ofrezca como mínimo métodos para hacer la autenticación, para realizar las consultas pertinentes de datos, para crear las transacciones y para validar un dato en concreto.

Además, se debe persistir al menos en la blockchain para tratar el **objetivo 2.1**: un resumen de los datos, una referencia para acceder a los datos en la base de datos externa, información para demostrar la autoría de los datos por parte del cliente e información para asegurar que esa transacción se hizo desde el *proxy*.

Por lo tanto, para tratar los **objetivo 4 y 5** se requiere realizar una implementación de los diseños que cubran estas necesidades para demostrar en un caso de uso práctico su funcionamiento. A su vez, será necesario implementar el software que hará de cliente de la plataforma para realizar un ejemplo de uso.

En resumen, de los objetivos planteados y en base a las necesidades del sistema surgen una serie de requerimientos:

- **Requerimiento 1:** Las transacciones persistidas en la plataforma deben ser confidencialidad, no debe quedar constancia clara del autor.
- **Requerimiento 2:** Los clientes que participan deben estar debidamente autenticados, demostrando con claridad a la plataforma información para identificarse. La autenticación puede delegarse en sistemas externos siguiendo un enfoque federado.
- **Requerimiento 3:** El diseño del proxy debe estar pensado para soportar el uso por parte de clientes IoT con capacidades limitadas.
- **Requerimiento 4:** Se debe asegurar la confidencialidad de los datos persistidos, para ello es necesario que se almacenen cifrados.
- **Requerimiento 5:** Los datos que se persistan deben poder ser validados para asegurar su integridad.
- **Requerimiento 6:** Se debe dejar constancia de meta-datos en las transacciones que otorguen información sobre la procedencia, por motivos de trazabilidad, de los datos persistidos.
- **Requerimiento 7:** La plataforma debe ser eficiente, para ello debe ser rápida y con transacciones ligeras (pocos datos y persistencia *offchain*).
- **Requerimiento 8:** Se debe asegurar la privacidad de los participantes y de sus transacciones, por ello no se puede dejar constancia de información que permita identificar fácilmente a los autores de las transacciones. Los meta-datos del sexto requerimiento deben ser los adecuados para respetar este requerimiento de privacidad sin perder la trazabilidad.

3.2 Propuesta de solución

El objetivo de la aplicación a desarrollar es el de servir de API para los dispositivos IoT, desde un enfoque proxy con el resto de tecnologías, de esto se puede ver que la API deberá ser simple, esto se podría lograr desarrollando un protocolo de mensajes sobre TCP muy básico para implementar la interfaz de funciones, este canal se podría proteger encriptando la comunicación de forma simétrica. Siguiendo con esta idea es posible hacer uso de estándares referentes a los protocolos de comunicación documentales como XML Schema, lo que facilitaría la definición del protocolo e incluso la implementación haciendo uso de herramientas automatizadas.

Otra opción que tiene mucha relación con XML Schema es la de hacer uso de uno de los estándares más importantes en la comunicación de procedimientos, SOAP, de tal forma que el desarrollo consistiría en realizar una API SOAP con mensajes definidos por XML Schema. Las tareas de encriptación y autenticación se deberían realizar de forma manual en la implementación de los métodos en el cliente y en la API ya que el protocolo de comunicación es independiente.

Por otro lado, otra alternativa muy interesante es la de implementar la API como un servicio REST, haciendo que toda la comunicación sea significativamente más simple y evitando la costosa gestión del protocolo SOAP y la validación de los XML.

Finalmente se escoge la opción de implementar una API REST por el motivo de que la tecnología REST ofrece una interacción basada en HTTP muy ligera y fácilmente utilizable en dispositivos limitados. Además, el uso de HTTP permite usar de forma sencilla las tecnologías desarrolladas para el mismo en nuestra aplicación, de tal forma que podemos resolver la necesidad de hacer el canal seguro simplemente haciendo uso de TLS en la conexión HTTP.

Respecto la autenticación del cliente, se han investigado y probado muchas alternativas de mecanismos de autenticación, los más interesantes han sido todos los que se basaban en el uso del certificado X.509 que tienen los clientes ya que este permite cífrados asimétricos, posee información para definir al cliente y está firmado por una entidad (CA) pública. El objetivo es que el cliente sea capaz de demostrar que es él al conectarse de forma segura y que el sistema pueda comprobar que es un cliente válido.

Para ello se diseña un protocolo simple de autenticación que, en primera instancia, hace uso de criptografía asimétrica para iniciar la comunicación y después usa de tokens generados por el servidor para autenticar el resto de la comunicación. Para validar el cliente se necesita que el certificado público esté dado de alta en la base de datos del proxy, una alternativa interesante que se ha estudiado es la de considerar a un cliente directamente válido si su certificado está firmado por una CA dada de alta en el sistema, de esta manera no es necesario “registrar” a los clientes en la plataforma.

Otra alternativa analizada es la de hacer uso de algún servicio externo de gestión de identidades (IdM), el interés práctico de esta opción ha sido tan grande que se ha decidido realizar el diseño de ambas alternativas, quedando separada la solución en dos esquemas, el esquema centralizado (el proxy gestiona las credenciales y la autenticación, es más simple) y el esquema federado (delega la autenticación en terceros, requiere más comunicaciones).

La idea del esquema federado consiste en hacer uso de un gestor de identidades, como Keyrock dentro marco IoT Fireware, y a su vez que este delegue la autenticación en un servicio que permita usar credenciales de ámbito nacional, como podría ser el sistema europeo eIDAS. Con esto se lograría desligar al sistema de la responsabilidad de la gestión de la información de credenciales, la aplicación únicamente guardará tokens válidos y toda la información referente a los clientes la

pedirá a eIDAS. En las **figuras 6 y 7** se muestran la arquitectura y el flujo de comunicación del Idm Keyrock con los nodos eIDAS.

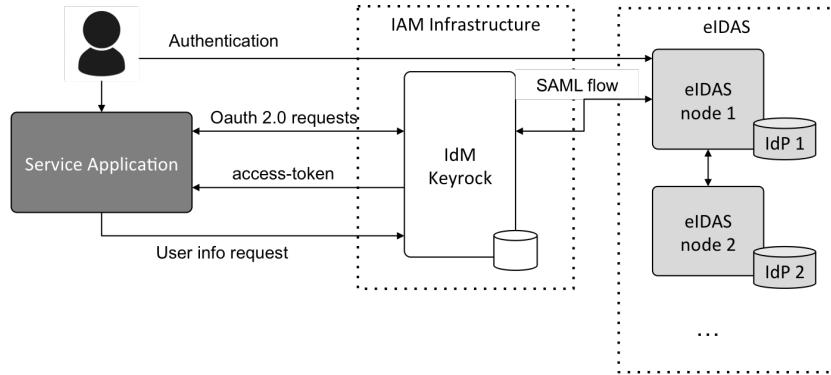


Figura 6: Arquitectura de la interacción de Keyrock con eIDAS, fuente [2]

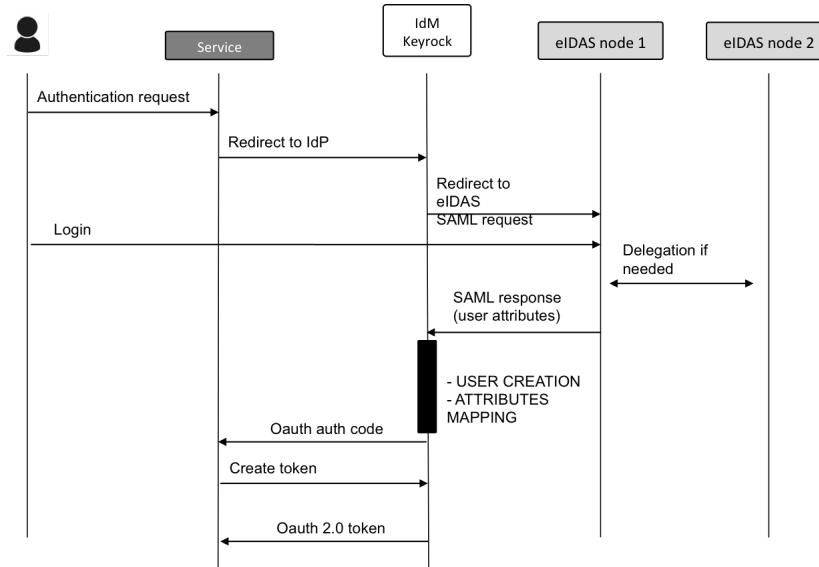


Figura 7: Flujo de comunicación de Keyrock con eIDAS, fuente [2]

Ambos esquemas, una vez realizada la autenticación respectiva, son iguales en el resto de pasos y en la implementación, los dos deberán realizar las transacciones y el envío de datos al IPFS. Con la propuesta de estas dos alternativas se consigue el **objetivo 3.1**.

Para la comunicación con la blockchain se dispone de dos alternativas, la primera consiste en que la aplicación implemente un cliente nativo de Hyperledger lo cual se descarta por su gran dificultad. Por otro lado, la segunda alternativa es mucho más factible y de gran interés, Hyperledger Composer dispone de muchas herramientas para facilitar a los programadores la interacción con Fabric, entre ellas ofrece un servicio REST que sirve de API HTTP para poder realizar todas las funcionalidades que proporciona la blockchain.

Por ello se considera que la aplicación deberá hacer uso de la API REST de Composer para comunicarse con Fabric, el proxy tendrá que ser cliente de esa API y será necesario que esta comunicación también sea segura y este autenticada para que solo puedan hacer uso del servicio de Composer clientes autorizados, entre ellos la aplicación desarrollada.

La API REST de Composer hace uso de diversas tecnologías de NPM, entre ellas dispone de “*Passport.js*” para la gestión modular del sistema de autenticación. Podemos usar los módulos de *Passport* para realizar la autenticación que más nos interese.

Hyperledger Composer, haciendo uso de una base de datos MongoDB, asocia el cliente autenticado a una *Card* que usará para firmar las transacciones, por ello podemos pensar que el método que más nos interesaría usar sería el de una autenticación basada en certificados (como se hace con el cliente). Tras implementarla vemos que en verdad la API de Composer está todavía en desarrollo por lo que muchos módulos de *Passport* no funcionan adecuadamente por lo que es necesario buscar otras alternativas.

Navegando entre las más de 500 estrategias de autenticación que tiene *Passport* vemos opciones muy básicas como podría ser el uso de un secreto compartido o de un usuario y contraseña, estas son descartadas por no ser muy seguras. Hay disponibles otras opciones federadas como usar OAuth con servicios externos (Google, Github, LinkedIn, etc..) que no son de nuestro interés al depender de servicio muy alejados del enfoque de la aplicación (en otros contextos estas opciones serían bastante útiles).

Finalmente encontramos un firme candidato, la estrategia basada en JWT (“*JSON Web Token*”), con ella el servicio REST de Composer simplemente requiere una clave privada que usará para verificar los tokens de los clientes, por otro lado, la aplicación desarrollada deberá autenticarse enviando el JSON con sus datos firmados con la clave del servidor (proceso que haremos manualmente al principio). Gracias a esto la API de Composer no aceptará conexiones que no tengan tokens JSON válidos por lo que su uso estará limitado a nuestra aplicación, además podemos hacer el canal seguro simplemente desplegando el servicio web sobre TLS.

Con todo lo dicho en los apartados anteriores tanto el canal de los clientes con el proxy como el canal del proxy con la blockchain serán seguros y las partes estarán debidamente autenticadas dando por cubierto por completo el **objetivo 3**.

Sobre el tema de guardar la información, como ya se dijo, se hará uso de un IPFS ya que no se recomienda guardar muchos datos en las transacciones de la blockchain. El sistema IPFS nos ofrece un mecanismo para referenciar los datos almacenados en él, este sistema se denomina CID (*Content Identifier - Identificador de contenido*) de forma que con una cadena de texto podemos identificar y localizar el archivo dentro del sistema IPFS. El sistema CID se basa en separar la cadena de texto en cinco partes, la primera de ellas representa el prefijo multibase (usado para indicar la base que se usa), la segunda parte muestra la versión de CID de esa cadena, la tercera parte describe como se codifica la siguiente parte haciendo uso de un multicodec, la cuarta parte describe el forma de hash haciendo uso de multihash para poder definir de forma dinámica cualquier función de resumen que se necesite, finalmente la quinta parte es el identificador del fichero como un resumen de la función indicada en el en la parte anterior. En la **figura 8** se muestra un ejemplo de CID indicando lo que representa cada parte del mismo.

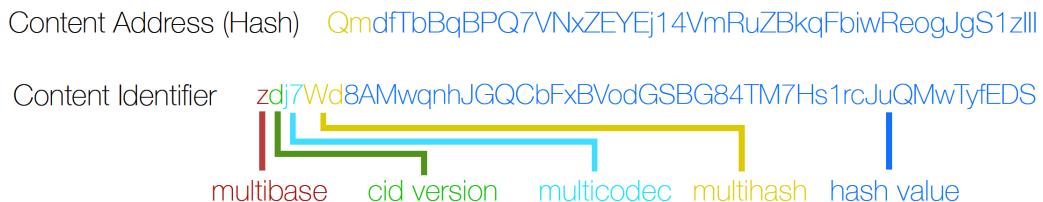


Figura 8: Descripción de la estructura del CID

En el diseño de la solución se hará uso del CID para poder guardar una referencia a los datos dentro de las transacciones. Para poder guardar un dato en la red IPFS se necesita enviar ese dato a un nodo que hará de semilla inicial, ese nodo será necesario que lo montemos nosotros.

La aplicación oficial de IPFS al iniciarla proporciona una interfaz web para gestionar el nodo, esta interfaz ofrece además un servicio REST para realizar las gestiones desde otras aplicaciones, para nosotros esto es muy útil ya que podremos enviar fácilmente los datos al nodo IPFS haciendo uso la API REST lo que mantendría el esquema general de servidor y cliente de APIs REST.

No es necesario cifrar y autenticar la conexión con el nodo IPFS ya que la información manejada en ese punto es de dominio público y va encriptada.

El proceso de encriptación de los datos recae en el cliente del sistema, de manera que él decidirá que método usará y con qué claves, se analizó la posibilidad de que la plataforma fuese la encargada del cifrado de los datos pero se abandonó la idea ya que implicaba una delicada gestión de claves por parte de la aplicación que no daba ningún tipo de flexibilidad, por otro lado, al relevar el cifrado al cliente se permite una total flexibilidad en los métodos y en la gestión de claves pudiendo seguir tanto enfoques de encriptación simétricos como asimétricos en función de las necesidades.

Entre las alternativas más interesantes para el cifrado está la de hacer uso de los certificados que usan los clientes al autenticarse, de los cuales son poseen la clave privada asociada, de forma que puedan cifrar los datos con su clave pública o la de otro cliente para asegurar que solo los dueños de esos certificados podrán descifrar los datos.

Además, se podría desarrollar paralelamente al *proxy* un servicio de gestión de credenciales que permita entornos de cifrado basados en atributos de forma que los clientes puedan realizar cifrados basados en los atributos que ellos cumplan logrando que, sin necesidad de intercambiar claves, otros clientes puedan desencriptar los datos si cumplen los mismos atributos.

En artículo “*Protecting personal data in IoT platform scenarios through encryption-based selective disclosure*”[26] se trata este tema en un contexto muy cercano al tratado en este documento, se analizan las alternativas de cifrados basados en políticas de atributos y el impacto que tendrían en entornos IoT junto con FIWARE.

Con lo dicho en estos párrafos se consigue satisfacer los **objetivos 2.1 y 2.2** respecto a la información que se deberá almacenar en la transacción y a la persistencia de los datos, gracias a esto también se da por completo el **objetivo 2** sobre el desarrollo de un mecanismo para realizar transacciones en la blockchain, y finalmente al haber tenido en cuenta el cifrado de los datos se da por cubierto el **objetivo 3.2**.

En la siguiente sección se explicarán los diseños de las soluciones finales que ofrecerán los mecanismos para realizar todas las tareas requeridas, por ello, se dará por cubierto el **objetivo 4** sobre el diseño completo de la solución final.

3.3 Diseño final

Tras las propuestas de soluciones se realizan dos diseños de aplicación, compartiendo ambos una estructura semejante ya que su principal diferencia está en el proceso de autenticación. En la **figura 9** se dispone de un esquema simplificado que muestra las tareas que agrupan los diseños.

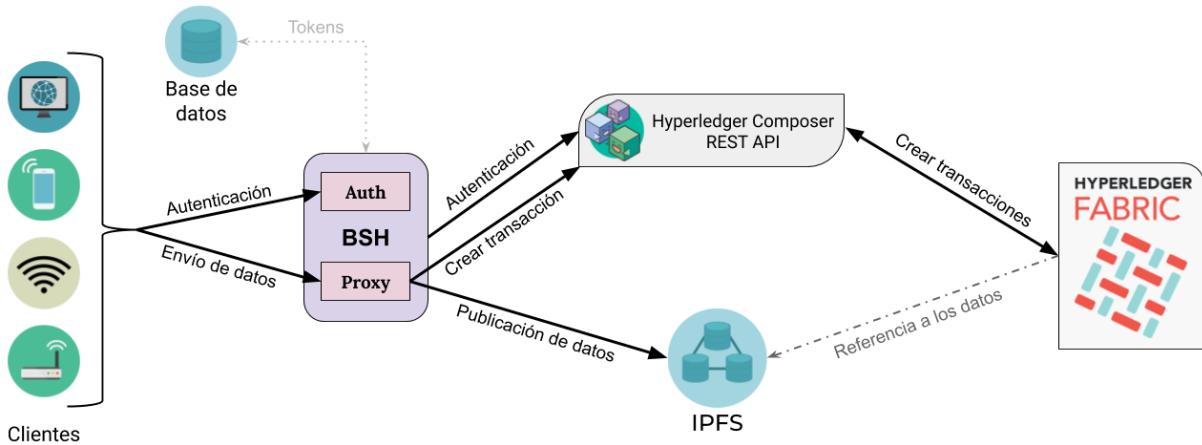


Figura 9: Esquema base del diseño final

Los dos diseños comienzan por tener en cuenta los clientes que puede tener el sistema, estos pueden ser de diferente naturaleza, desde dispositivos IoT hasta equipos personales pasando por móviles. Los clientes accederán a la aplicación haciendo uso de una interfaz REST sostenida por el estándar HTTP junto con TLS para hacer seguro el canal de comunicación.

La plataforma desarrollada se nombra **BSH** (“*Blockchain Security Handler*”) y está formada por dos módulos funcionales, separados conceptualmente, que realizan sus tareas de forma aislada.

Por un lado, el primer módulo, denominado “*Auth*”, se encarga de realizar el protocolo de autenticación y la gestión de credenciales de los clientes, el comportamiento de este módulo será bastante distinto en cada esquema.

Por otro lado, el segundo módulo, denominado “*REST*” y en algunos contextos “*Proxy*”, es el encargado de realizar todas las operaciones propias del sistema, es el que ofrecerá los mecanismos de recepción de datos y petición de información. Este módulo es el que realiza el *proxy* en sí con la blockchain sirviendo de puente entre el cliente e Hyperledger.

Ambos módulos comparten una base de datos que les será necesaria para almacenar la información de los clientes y para el protocolo de autenticación, entre esa información la más relevante es el token de sesión de los clientes autenticados.

Respecto al funcionamiento, el diseño requiere de un proceso inicial de autenticación del cliente frente al módulo de autenticación, si se logra correctamente la aplicación le devolverá un token de sesión que deberá usar posteriormente para realizar todas las comunicaciones con el otro módulo de forma autenticada.

Tanto el cliente como el servidor harán uso del formato JSON para estructurar los datos en las comunicaciones. El cliente pondrá siempre un campo en los mensajes donde se indique su token de sesión, el resto de campos varían según el tipo de función que se esté realizando.

La aplicación se comunicará con la API REST de Composer de forma autenticada (mediante JWT) y con mensajes también en formato JSON, con una estructura parecida a la que ofrece el servicio proxy desarrollado.

Además, la comunicación con el nodo IPFS será, como se dijo, mediante la API REST que ofrece la aplicación oficial, aunque en la implementación (como se verá posteriormente) no se hará de forma directa.

En el diseño también es necesario definir la funcionalidad principal que ofrecerá cada módulo de la plataforma a los clientes que harán uso de ella.

El módulo de autenticación proporcionará a los clientes la funcionalidad necesaria para que puedan autenticarse (obtener un token de sesión), para probar si un token es válido y para borrar un token (cerrar sesión).

El módulo Proxy ofrecerá métodos para realizar las tareas de gestión de las transacciones. Entre ellas, se permitirá realizar el envío de datos para crear transacciones y se ofrecerán mecanismos para realizar tanto consultas de listados completos de transacciones como de unas en concreto en función de unos determinados parámetros. Además, deberá tener operaciones para realizar la verificación de un dato concreto frente a la blockchain y para la obtención de determinados documentos almacenados en el IPFS.

Finalmente, en la **figura 10** se muestra un esquema simplificado del *Blockchain Security Handler* desarrollado, mostrando los componentes que lo forman, los sistemas de los que depende y la funcionalidad que ofrece.

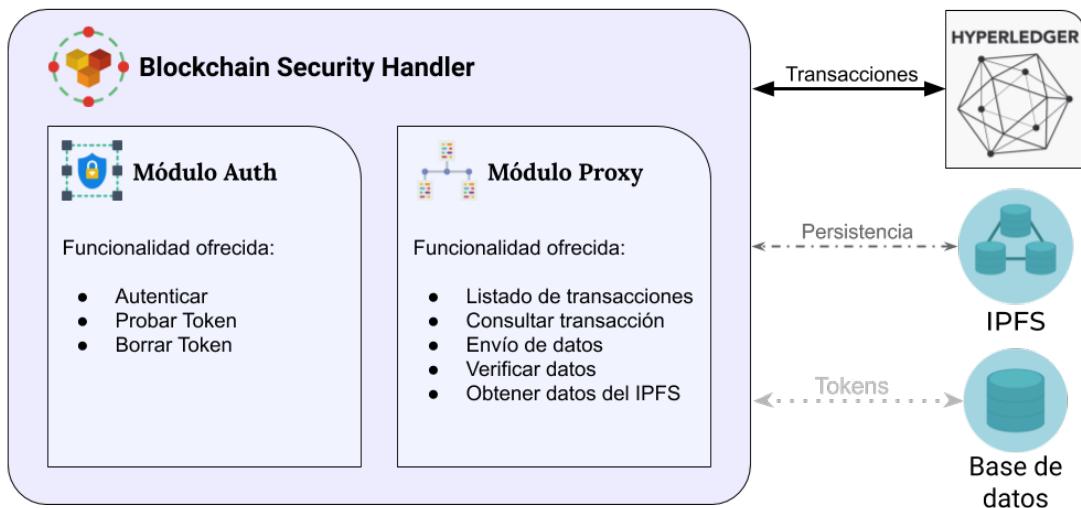


Figura 10: Esquema simplificado de los componentes del *Blockchain Security Handler*

3.3.1 Diseño del esquema centralizado

El primer diseño de la solución propuesta nos lleva a este esquema, en el cual el Blockchain Security Handler tiene la responsabilidad de gestionar las credenciales e información de identidad de los clientes y también es el encargado de realizar el proceso de autenticación de los mismo. Para almacenar la información le será necesario usar la base de datos de la que dispone donde guardará los certificados de los clientes dados de alta en el sistema.

Dado que en este esquema el sistema es el encargado de gestionar las credenciales se hace necesario un proceso por el cual se den de alta a los clientes que se quieran aceptar en el sistema, este proceso será manual ya que el administrador del sistema deberá meter el certificado en la base de datos.

Para reducir los inconvenientes anteriores y ganar en escalabilidad se podría hacer uso de otra alternativa que facilitaría la adición de nuevos clientes, esta consiste en dar de alta en el sistema a entidades certificadores en las cuales confiemos de forma que estas sean las que expidan los certificados de los clientes. Así pues, cuando sea necesario validar el certificado de un cliente no hará falta comprobar si está dado de alta en el sistema, será suficiente con comprobar si es un certificado válido y si está firmado por una CA confiable para la plataforma.

En la **figura 11** se muestra un esquema del diseño centralizado en la cual podemos ver los componentes que forman parte de la misma. También podemos ver la comunicación detallada de cada componente en la **figura 12**, en la cual se muestra el flujo de comunicaciones del diseño.

Este esquema sigue el siguiente modo de funcionamiento:

1. Como ya hemos dicho, es necesario autenticarse, para ello el cliente hace uso de su certificado X.509 enviándoselo al módulo de autenticación, pero no prueba que posee la clave privada del mismo ya que simplemente envía el certificado público
2. El módulo de autenticación comprueba que el certificado sea válido (esté bien formado y no haya expirado) y después comprueba si está dado de alta en el sistema o si proviene de una CA confiable, si ambos comprobaciones salen bien se procede a la generación de una cadena de texto aleatoria que servirá de token de sesión y este se guardará en la base de datos asociándolo al cliente y marcándolo en un estado temporal, de forma que tendrá un tiempo de vida muy corto para evitar crear muchos token que no se acaben usando (clientes que no logran autenticarse), finalmente se envía el token encriptado con la clave pública del certificado al cliente.
3. El cliente recibe el token encriptado, en este punto es donde radica la autenticación ya que si el cliente es poseedor de la clave privada del certificado podrá desencriptar el token, en caso contrario no podrá por lo que no tendrá ningún token válido. Tras desencriptar el token adecuadamente este se guarda ya que se usará en toda la comunicación como token de sesión.
4. Cuando tenemos una sesión creada se procede a realizar el envío de los datos (el ejemplo que se está estudiando es el más complejo, los otros métodos tendrán un funcionamiento parecido), para ello se empaqueta el token de sesión, los datos y los meta-datos en un mensaje JSON que se enviará al módulo REST de mi aplicación.
5. Cuando el módulo REST recibe el mensaje comprueba si el token es válido, además comprobará si es un token en estado temporal (es la primera vez que se usa) y lo actualizará a un estado en uso (tiempo de vida mucho más largo). Tras eso, desempaquetará el mensaje

y envía el dato al nodo IPFS haciendo uso de la interfaz REST que nos proporciona la aplicación oficial.

6. Se espera a que el IPFS devuelva el CID del fichero, cuando se recibe se guarda para el siguiente paso.
7. Con toda la información lista y calculada (meta-datos adicionales que sean necesarios) se procede a generar la transacción en la blockchain, para ello se empaquetan la información que queremos persistir junto con el CID que referencia al dato recibido en un JSON con la estructura que nos requiere Hyperledger, tras eso se envía a la API REST de Composer para que ella se conecte con Fabric y cree la transacción. Hay que recordar que la aplicación debe estar autenticada frente a la API REST de Composer, esto lo hace en el momento en el que se inicia el Blockchain Security Handler con un simple mensaje JSON donde se le pasa el identificador JWT firmado con la clave privada que le dimos a la API de Composer.
8. Se recibe el resultado de la transacción donde tenemos el modelo que definimos con todos los datos llenados, entre ellos el identificador de transacción y la marca de tiempo.
9. Finalmente se reestructura el resultado del paso anterior al formato que hemos definido para manejar esos datos y se le envía al cliente.

Con todo esto tendríamos un esquema autosuficiente de la propuesta de solución centralizada que podría ser factible de ser implementado para realizar el caso práctico de ejemplo.

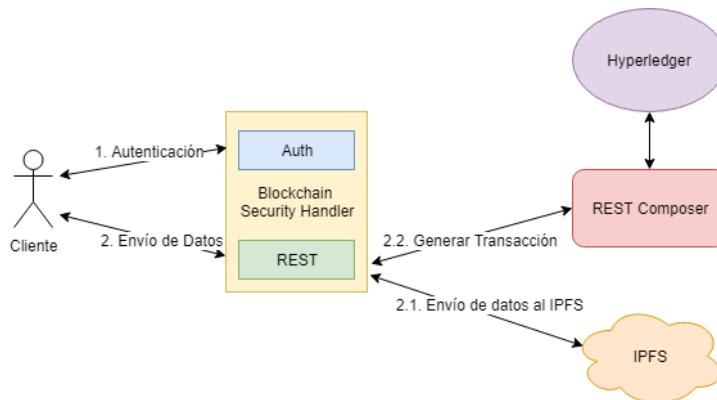


Figura 11: Esquema del diseño centralizado

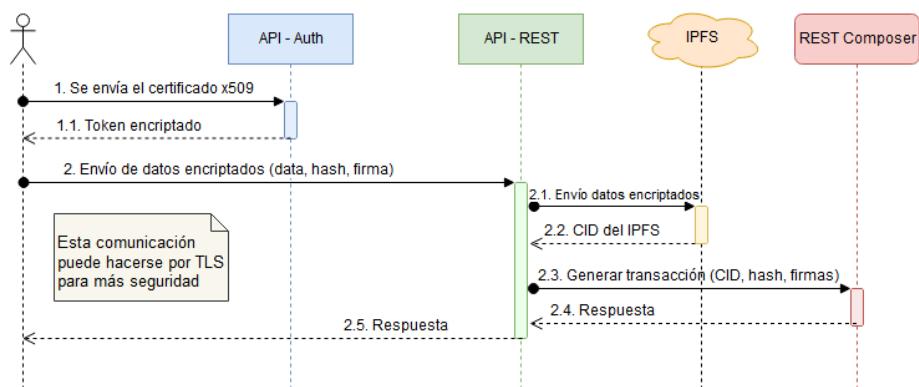


Figura 12: Flujo de comunicaciones del diseño centralizado

3.3.2 Diseño del esquema federado

Alternativamente disponemos de otro esquema en el cual se delega el proceso de autenticación y la gestión de credenciales en sistemas externos que proporcionan interfaces para realizarlo.

Como ya se dijo, el proceso de autenticación se cede a un gestor de identidades que nosotros mismo podemos desplegar, en este caso se hace uso del proporcionado por la plataforma IoT Fiware (Keyrock), este nos permite usar el estándar eIDAS para realizar una autenticación con credenciales europeas delegando este proceso a la red eIDAS.

Con este esquema el *proxy* no tiene que gestionar ningún tipo de credenciales o información de los clientes, quitando esa responsabilidad y eliminando las necesidades de seguridad en el almacenamiento de esos datos. A su vez, es un sistema interesante ya que los clientes no deben darse de alta en la aplicación y se puede ofrecer directamente el acceso a una gran red de usuarios europeos de forma sencilla y con total seguridad legal en el uso y verificación de las identidades.

Tanto el esquema como el flujo de funcionamiento de este diseño es muy similar al anterior, cambiará todo lo referente al módulo de autenticación. En la **figura 13** se muestra el esquema del diseño federado donde se aprecian las dos nuevas entidades. También se pueden ver las nuevas comunicaciones en la **figura 14**, en la cual se muestra el flujo de comunicaciones del diseño.

El nuevo esquema sigue el modo de funcionamiento explicado a continuación:

1. El cliente realiza una petición de autenticación al módulo de autenticación, este le redirecciona al IdM Keyrock el cual, al estar configurado para hacer uso de eIDAS, vuelve a redirigir al cliente a un nodo eIDAS. Keyrock procederá a realizar una petición SAML a ese nodo.
2. El cliente se autentica en el nodo eIDAS usando su certificado X.509 asociado a una identidad electrónica legal, por ejemplo, usando el DNIe o un certificado de la FNMT.
3. Tras eso el nodo le responde a Keyrock con una respuesta SAML para probar la correcta autenticación del cliente.
4. Keyrock responde al módulo de autenticación con un código de autenticación del estándar OAuth 2.0.
5. Con el código el módulo de autenticación procede a pedir a Keyrock la creación de un token de sesión con él.
6. Keyrock responde con el token OAuth 2.0 para la sesión de ese usuario. Si fuese necesaria alguna información del cliente se pediría a eIDAS los datos del dueño del certificado.
7. Finalmente el módulo de autenticación crea un token de sesión como el esquema anterior (estado temporal) y se lo envía al cliente sin encriptar ya que está probada la identidad del cliente.
8. El cliente usa ese token de sesión en el resto de comunicaciones con la aplicación.
9. El resto de pasos son los mismos que el esquema centralizado.

Aun siendo un diseño muy prometedor, gracias a su gran interoperabilidad entre gestores de identidad, en la resolución práctica no se usará. Esto es debido a que actualmente Keyrock tiene una importante limitación en los mecanismos de autenticación con su plataforma, a raíz

de una actualización en su API se eliminaron los mecanismos de autenticación con certificados y se sustituyeron por mecanismos de usuario y contraseña (a través de un formulario web) lo que hace directamente imposible cumplir las especificaciones de la solución que se quiere desarrollar.

Además, se estudió el mecanismo de interacción con eIDAS para poder desplegar una comunicación con un nodo español oficial pero también se vio inviable por ser demasiado complejo de realizar ya que había unos requisitos legales y técnicos que no se podían lograr a tiempo para las pruebas.

En el caso de España, la integración con el sistema eIDAS se realiza a través de la plataforma común del Sector Público Administrativo Estatal **Cl@ve Identificación**.

Cl@ve es un sistema orientado a unificar y simplificar el acceso electrónico de los ciudadanos a los servicios públicos, se trata de una plataforma interoperable común para la identificación, autenticación y firma electrónica que evita a las Administraciones Públicas tener que implementar y gestionar sus propios sistemas de identificación y firma, y a los ciudadanos tener que utilizar métodos de identificación diferentes para cada una. **Cl@ve** contempla la utilización de sistemas de identificación basados en claves (sistemas de usuario y contraseña) y certificados electrónicos (certificados expedidos por la *FNMT* o los del *DNIe*).

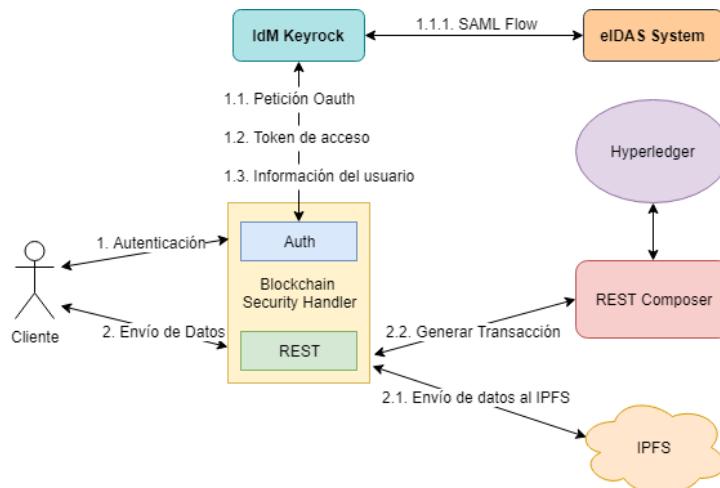


Figura 13: Esquema del diseño federado

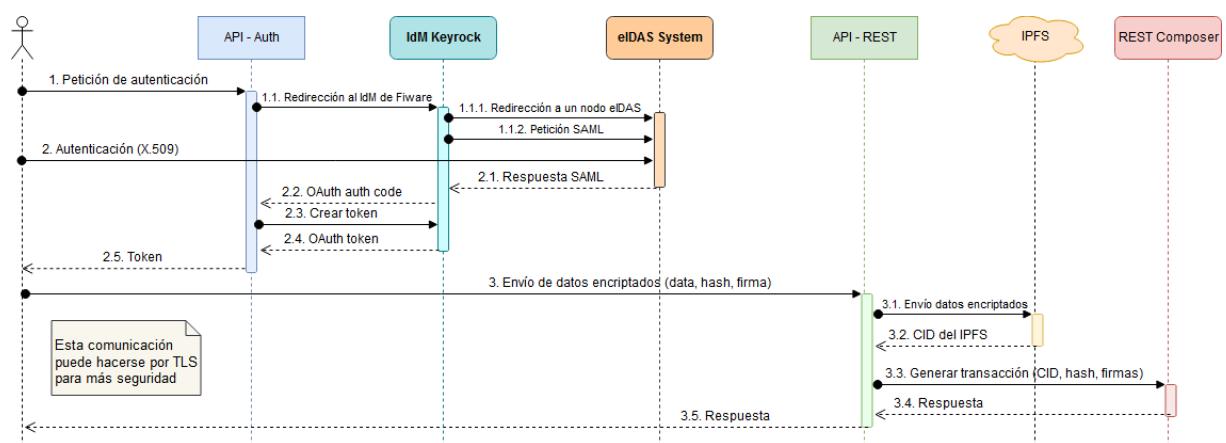


Figura 14: Flujo de comunicaciones del diseño federado

3.4 Diseño de clientes

Respecto a los usuarios del sistema, es posible separarlos en dos tipos principales de potenciales clientes, por un lado tendríamos los dispositivos personales y por otro lado estarían los dispositivos IoT.

Como se mostró en la **figura 5**, el mayor interés de los clientes de la plataforma es el de persistir datos para que sean auditables y para dejar constancia de su creación de forma segura. Para ellos la solución que se ha desarrollado será como una base de datos segura, distribuida e inmutable en la que podrán guardar documentos manteniéndolos privados pero de forma que cualquier miembro pueda verificar cuando se subió el documento al sistema y verificar la integridad del mismo.

A continuación, se va a separar el diseño de los clientes en función de los dispositivos que usen, en ambos casos se diseñará además un ejemplo práctico de uso del sistema desarrollado, sirviendo de base para la implementación del caso de uso.

3.4.1 Dispositivo Personal

Los dispositivos personales son todos aquellos que tengan un carácter de uso general por un individuo en concreto, podrían ser móviles, tabletas, portátiles y equipos. Las necesidades de estos clientes son las de persistir documentos personales de forma segura, por ello, el tiempo que se tarde en realizar este proceso no es una prioridad.

Estos clientes disponen de un certificado de identidad X.509 propio que usarán para autenticarse. Se desarrollará una aplicación web para demostrar un escenario de persistencia de documentos electrónicos a modo de PoC (*Proof of Concept - Prueba de concepto*). El objetivo es exemplificar la utilidad del sistema para la certificación de documentos electrónicos por si fuera necesario, por motivos legales, acreditar la creación del mismo en un determinado momento. A su vez, la plataforma podría servir de servicio de alojamiento de documentos.

Un ejemplo particular es el mostrado en la **figura 15**, en este un usuario desea entregar un documento a otra persona asegurando que solo ella lo podrá leer, además, le interesa dejar constancia del momento exacto del envío de ese fichero así como de los datos en sí.

Para lograr esto, el usuario deberá hacer uso de la aplicación como mediador confiable en la transmisión del documento.

Lo primero que tiene que hacer el usuario es autenticarse en el sistema, tras eso le pedirá, si no lo tuviese ya, el certificado público del usuario al que se quiere enviar el documento. Posteriormente creará el documento con los datos que quiere enviar, lo firmará con su certificado personal y lo encriptará con la clave pública del certificado del receptor. Finalmente enviará el fichero al sistema para que se registre adecuadamente.

El usuario que tiene que recibir el documento tiene varias maneras de obtenerlo, el usuario que creó el documento se lo puede enviar directamente (encriptado o no) y simplemente usar la plataforma como base de datos, por otro lado, la forma más idónea para hacer el uso completo de la infraestructura desarrollada es que se le pase el identificador de la transacción en el sistema, con ella el receptor deberá pedirle al sistema todos los detalles de la misma y entre ellos estará el CID del servicio IPFS que le servirá para descargar el documento encriptado.

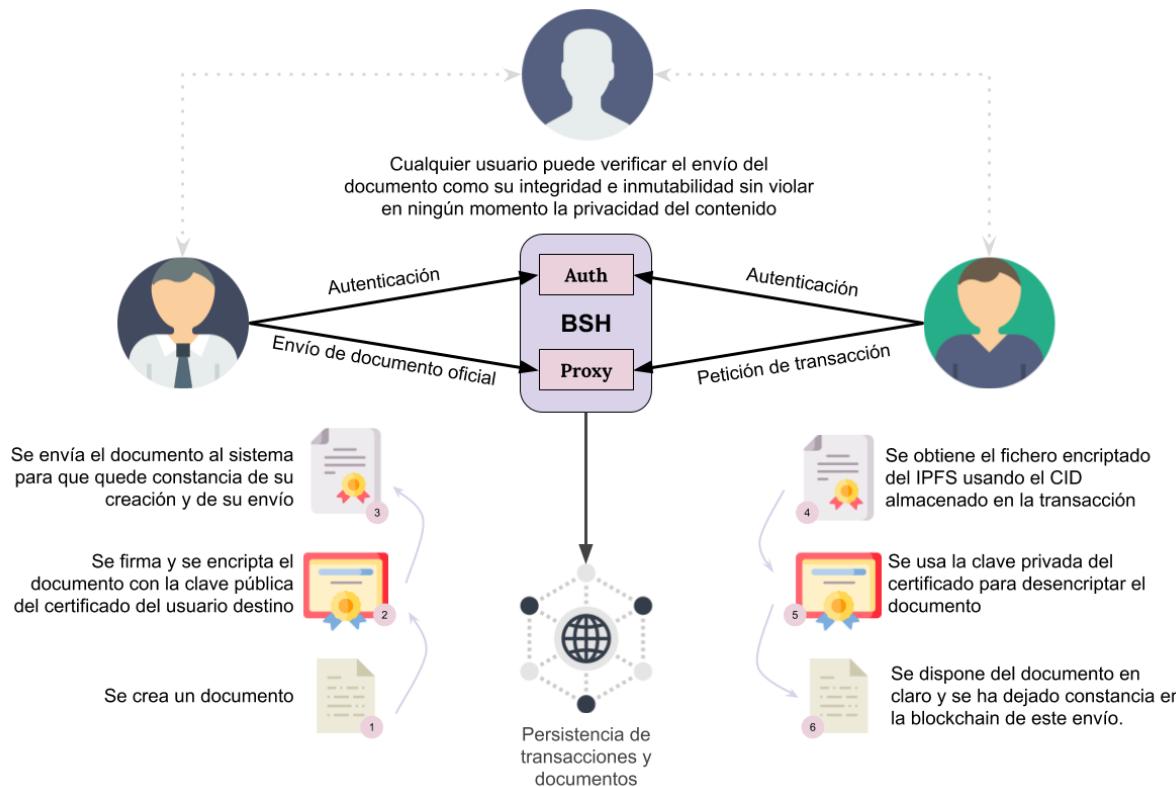


Figura 15: Ejemplo de escenario de persistencia de documentos oficiales

Con el documento descargado procederá a desencriptarlo haciendo uso de la clave privada de su certificado, solo él podrá hacer este proceso lo que asegura la privacidad de esos datos, obteniendo tras esto el documento en claro. Tras eso, podrá verificar la firma del productor y la integridad del documento asegurando que solo él ha podido leerlo y que vienen del auténtico dueño, todo esto dejando constancia pública del envío del documento y del contenido del mismo (encriptado y resumido).

También sería posible hacer el proceso de encriptación al contrario, un usuario podría publicar en el sistema un documento firmado con su clave privada de forma que cualquier usuario que tenga acceso a la plataforma pueda pedir su certificado público y descargar el fichero, si logra desencriptar el documento adecuadamente podrá asegurar que el documento fue creado y encriptado por el dueño del certificado.

Incluso se podría llevar al extremo, el usuario que crea el documento lo firma con su clave pública de forma que solo él podrá ver el contenido, asegurando una total privacidad del documento. En caso de que tenga que justificar legalmente ese documento podrá usar la plataforma para asegurar el momento en el que lo creó y podrá hacer uso de su certificado (con validez legal) para desencriptar el fichero y mostrar el contenido.

Un ejemplo de este uso podría ser el de registrar legalmente el código fuente de una aplicación privada, si un usuario sube el código encriptado nadie lo podrá leer pero si por algún motivo sufre un plagio, ante un juez podría demostrar que el autor original fue él ya que tendría una transacción que avala la fecha de creación y además podría desencriptar el contenido justificando la autoría del código.

3.4.2 Dispositivo IoT

Los dispositivos IoT representan una fuente de datos importante aun disponiendo de características bastante reducidas, su principal necesidad es lograr persistir en la blockchain los datos resumidos que recogen de la forma más rápida posible ya que están continuamente produciéndolos.

Ya que queremos hacer una PoC para probar nuestra solución, será necesario desarrollar un cliente para un SBC (*Single-Board Computer*) que simulará el comportamiento de un dispositivo IoT, para ello se hará uso de las populares Raspberry. Este cliente simulará una aplicación que produzca un determinado dato y que deba ser almacenado en nuestro sistema de forma periódica.

La placa que se usará será una versión lo más reducida posible de la Raspberry, esta es la Raspberry Pi Zero W la cual nos proporciona una importante potencia teniendo en cuenta el tamaño que tiene (ver **figura 16**) y su reducido coste. La placa a su vez integra WiFi y Bluetooth lo que da muchísimas posibilidades a la hora de desarrollar dispositivos IoT.

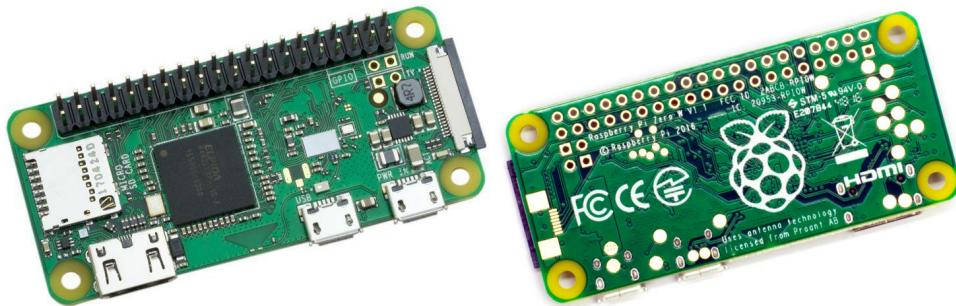


Figura 16: Raspberry Pi Zero W

El equipo de Raspberries proporciona una distribución de Linux modificada denominada Raspbian para instalar en las Raspberries, aun así, toda su familia de dispositivos se construye sobre procesadores ARM por lo que es posible instalarles una amplia variedad de sistemas operativos (la gran mayoría son distribuciones de Linux).

Las ventajas de tener un sistema operativo completamente funcional como puede ser Linux puro es que se puede desarrollar sin ningún tipo de limitación ya que se tiene un soporte completo de las tecnologías y lenguajes de programación actuales, además las aplicaciones desarrolladas son fácilmente portables a otros sistemas sin muchas dificultades.

Otra cosa muy ventajosa de disponer de un sistema operativo es que partimos ya de un sistema que tiene implementada y gestiona automáticamente las entradas y salidas, disponemos de Ethernet directamente por lo que solo tenemos que desplegar un Socket básico en cualquier lenguaje de programación. Por otro lado, podemos hacer uso de la multitud de librerías y paquetes desarrollados para Linux desde los más básicos hasta interfaces completas para el mantenimiento y gestión del dispositivo.

Nuestro objetivo con este dispositivo es diseñar un esquema de uso para el ejemplo práctico de la siguiente sección. Se deberá desarrollar un cliente que haga uso en la autenticación de un certificado que simulará el certificado que traen de fábrica los dispositivos IoT típicamente.

El ejemplo consistirá en autenticarse en el *proxy* y tras eso realizar una serie de envíos con

datos simulando el funcionamiento de un sensor de una determinada máquina en una planta de producción.

El sensor captará de la máquina varias propiedades, medirá el nivel del tanque que tiene, registrará la temperatura de la máquina y comprobará la presión de los dispositivos hidráulicos. Toda esta información deberá ser empaquetada en un único paquete que, dada la importancia de esta máquina y la necesidad de dejar registrado todo lo que hace, deberá ser enviado cada minuto al sistema para que se persista en la blockchain.

En la **figura 17** se muestra conceptualmente el ejemplo de uso descrito.

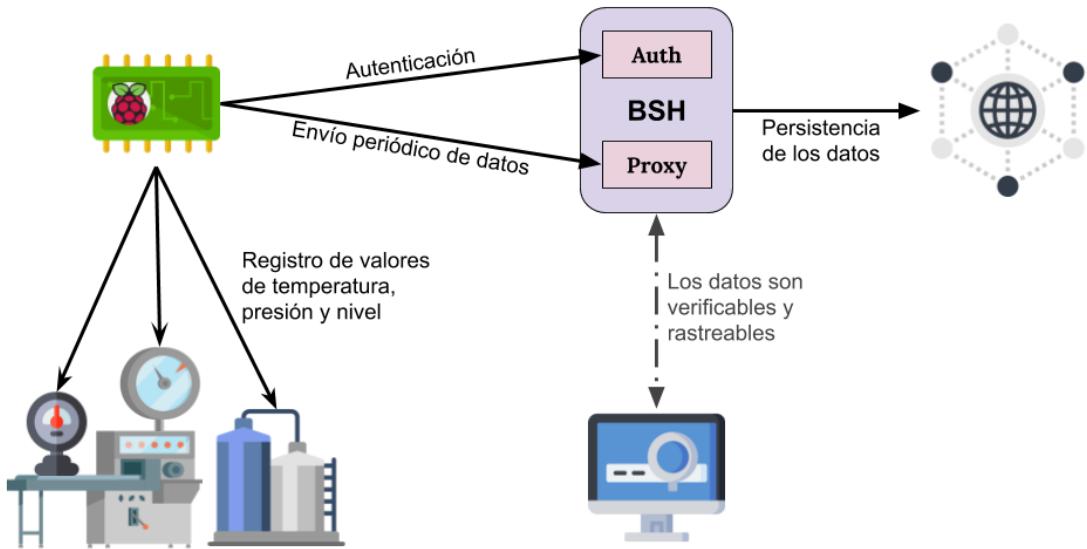


Figura 17: Ejemplo de escenario IoT

Existen otros ejemplos de uso interesantes dentro del mismo contexto, por ejemplo, se podría registrar en la blockchain el tránsito de personal en una zona restringida, o el número de productos resultantes de un lote, o cualquier tipo de información que proporcione el sistema SCADA de la fábrica y que sea de interés el almacenarlo en un sistema público.

Hay que recordar que no hace falta que la plataforma *proxy* junto a la blockchain formen parte de la empresa que hace uso de ella ya que esta intenta ser un sistema interoperable que ofrezca un punto de acceso común entre varias entidades. Se podría considerar la plataforma como un servicio externo en el que participaran diversas empresas que tienen en común la necesidad de conocer entre ellas cierta información.

4 Implementación del caso de uso

En esta sección se realizará la implementación de uno de los diseños de solución propuesto en el apartado anterior. Dado que en la sección anterior ya se han analizado las alternativas y se ha explicado el diseño propuesta, este punto tratará todo lo referente a la implementación software y el despliegue en sí, teniendo en cuenta todos los detalles propios de la realización práctica de la aplicación. Finalizada esta sección se dará por cubierto el **objetivo 5** referente al desarrollo de la solución.

Junto a la realización de la aplicación *proxy* se realizarán tres softwares clientes para servir de ejemplo de uso del sistema.

4.1 Blockchain

La tecnología blockchain que se usará será Hyperledger Fabric, además, se desplegará siguiendo la modalidad de virtualización en la que se prefiere que todo esté alojado en un único equipo, de carácter personal o dedicado, el cual ejecutará una máquina virtual en el hipervisor Virtualbox la cual estará gestionada mediante Vagrant.

La máquina virtual (*VM*) contendrá el software de Hyperledger, en ella se desplegarán los programas necesarios por medio de contenedores Docker, de forma que se podrán crear virtualmente en la misma máquina todos los nodos que necesita la red para funcionar.

Por otro lado, el host estará debidamente configurado mediante variables de entorno para que sea capaz de comunicarse con la máquina virtual y poder gestionar los contenedores de Docker y la conexión con Composer si tener que acceder al equipo virtual. En el primer anexo se explica cómo realizar el proceso de instalación y configuración de la plataforma Fabric.

En la **figura 18** se muestra de forma simplificada el esquema de despliegue junto con los contenedores y máquinas que se ejecutarán.

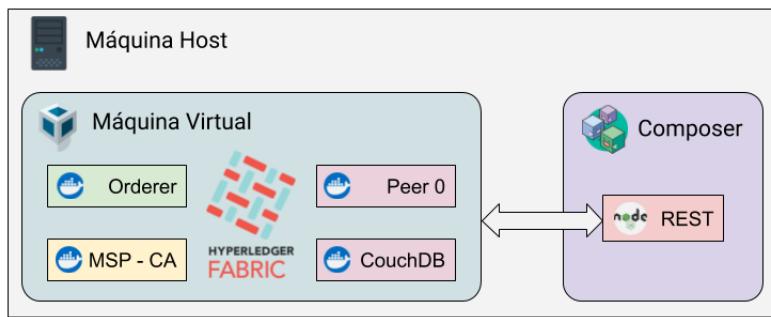


Figura 18: Esquema de despliegue virtualizado de Hyperledger Fabric

Cuando se dispone de la blockchain Fabric operativa se puede avanzar al siguiente paso, la creación de la aplicación que modelará las transacciones del sistema. Para lograrlo se debe usar Yeoman para generar un esqueleto de aplicación y Composer para realizar fácilmente toda la comunicación con Fabric, en el segundo anexo se explica este proceso.

Como ya se dijo, en la terminología de Hyperledger a las aplicaciones que ejecuta Fabric se les denomina *Business Network Application (BNA)* por su carácter distribuido y empresarial.

Dentro del esquema de BNA proporcionado por Yeoman se disponen de diversos directorios y ficheros para definir características de la aplicación, el que interesa para la implementación de la solución son los contenidos en la carpeta “models”, en ella estarán los ficheros que modelan los “Assets” de la aplicación.

La BNA necesaria para cumplir los esquemas propuestos tendrá un único fichero de modelo que contendrá la definición de una transacción. En esta transacción se almacenarán varios campos de información adicional, sumándose a la marca de tiempo y a la id de la transacción.

La información adicional necesaria para cumplir las especificaciones de la plataforma será:

- **CID:** Será necesario almacenar la cadena de texto (CID) que proporciona el sistema IPFS al publicar un fichero para que se pueda enlazar una transacción con los datos que persistió.
- **Hash:** Para que sea posible demostrar la integridad de los datos deberá guardarse un resumen (usando la función que se vea necesaria) de la información registrada.
- **Firma de usuario:** Como se dijo, es necesario que junto a los datos de la transacción vaya información para asegurar y certificar quien produjo esa información, esto se logra añadiendo un campo donde se guarde una firma. Para realizar la firma el usuario hace uso de su clave privada para encriptar el hash, así cualquier otro usuario podrá comprobar fácilmente (usando la clave pública) la veracidad de la procedencia.
- **Firma del Proxy:** Analoga a la firma del usuario, el sistema hace uso de su clave privada para firmar tambien el resumen, de esta forma se añade una capa extra de seguridad ya que aseguramos que la transacción ha sido creada en la plataforma.

El fichero que contiene el modelo, configurado según los parámetros explicados y siguiendo la sintaxis de Composer, quedaría así:

```

1 namespace iot
2
3 transaction IoTTransaction {
4     o String cid
5     o String hash
6     o String firmaAPI
7     o String firmaUser
8 }
```

Se tendría entonces todo lo necesario para persistir en la blockchain la información y para cumplir los objetivos de la plataforma. Se estudió la posibilidad de meter más parámetros relacionados con el productor de los datos (identificador, descripción de los datos, tags, métodos de cifrado o firma, etc..), pero finalmente no se pusieron e incluso se limitó al máximo la información que se almacena en la transacción.

El motivo de reducir los parámetros de las transacciones es provocado por dos factores. Por una lado se tiene en cuenta la escalabilidad, ya que la blockchain tendrá que almacenar una gran cantidad de transacciones e interesa que tengan el menor tamaño posible.

Por otro lado, un motivo muy importante para no poner datos que identificasen directamente al productor es el de evitar la trazabilidad de los datos con el cliente que los produce, esto limitaría la privacidad de la información que se gestiona en la plataforma, por ello interesa que difícilmente se pueda saber quién es el autor de los datos solo con la transacción.

Esta limitación no afecta a la auditabilidad y verificabilidad de los datos ya que cualquier usuario podría proponerse públicamente como autor de los datos, mostrando su certificado junto con una

prueba de que dispone de la clave privada, y todos los demás podrían verificar que efectivamente es el productor simplemente comprobando la firma de los datos.

Finalmente queda realizar el proceso de compilación de la BNA y su publicación en la blockchain, para ello se hace uso de las herramientas que proporcionan Hyperledger Composer. Este proceso se explica en la parte final del segundo anexo.

Tras tener completamente funcional la blockchain y la BNA es necesario montar el servicio REST que ofrece Composer para servir de interconexión entre la aplicación a desarrollar y Fabric.

Aunque el proceso para montar la API de Composer se detalla en el tercer anexo se pueden comentar algunos aspectos relevantes de la misma. El despliegue del servicio REST se hará en la máquina host mediante NodeJS, además, es necesario realizar una configuración concreta de las opciones de lanzamiento para que haga uso de *passport.js* con la estrategia JWT, indicándole la clave privada que usará para firmar, y que deberá usar una base de datos MongoDB para guardar las credenciales de los usuarios que acceden.

Por lo tanto, para que el servicio proxy pueda hacer uso de la API REST de Composer necesita un token JWT válido, para ello se puede hacer uso de la página **jwt.io** para facilitar este proceso.

En la página solamente es necesario llenar el campo *payload*, que representa los datos del cliente, y el campo de la clave privada, tras eso la página mostrará en el lado izquierdo de forma gráfica el JWT que se tiene que usar al autenticar el proxy.

En el campo *payload* se puede modelar la información siguiendo el formato JSON, en el caso de la solución propuesta se usan dos campos, uno con una marca de tiempo y otro con un identificador de usuario, en un principio se puede añadir cualquier información que se vea relevante con tal de identificar correctamente al cliente.

La ventaja de JWT es que permite una total flexibilidad en el contenido del campo payload ya que cada cliente puede almacenar ahí lo que considere necesario.

Como último detalle referente a la configuración de *passport.js* con JWT, en la implementación se realiza por comunidad el paso del JWT como un parámetro GET en la ruta, la estrategia original obtiene el token desde un parámetro de la cabecera.

Para cambiar la manera en la que se obtiene el token de sesión se requiere modificar la estrategia para sobrescribir la función que busca el token por una que lo extraiga de la ruta. La configuración necesaria se mostrará en el tercer anexo.

4.2 Blockchain Security Handler

La plataforma a desarrollar es, en la práctica, una versión levemente simplificada del esquema centralizado de las soluciones propuestas, esto ocurre porque no se aspira a la creación del sistema completamente funcional para ser usado en producción, por el contrario, se prefiere crear una aplicación que sirva de ejemplo para demostrar el funcionamiento básico de la propuesta.

La aplicación se implementará en el lenguaje **Java**, en la edición *Enterprise*, siguiendo el estándar Java-RS para desarrollar una plataforma REST. Se usará la librería **Jersey** como implementación de referencia de Java-RS y **Apache Tomcat** como servidor de despliegue.

El proxy se realiza en un proyecto *Java Web* en el entorno **Eclipse** donde se agrupará todo el software desarrollado separado en paquetes. Los paquetes que contiene el proyecto son los siguientes:

- **alice:** Este paquete contiene el código de la librería *alice* para realizar cifrados en AES/DES. El código está disponible en el repositorio oficial (github.com/rockaport/alice).
- **modelo:** Aquí se almacenan las distintas clases que servirán para modelar las entidades que se manejan en la aplicación
- **rest:** Este es el paquete que analizará Jersey en busca de clases destinadas a ser servicios REST, en el caso del proxy habrá una clase que contendrá todas las operaciones que ofrece la plataforma.
- **stub:** Paquete para clases “*stub*”, para realizar pruebas se crean clases simplificadas que simulan el comportamiento de la original. La única que hay disponible es una versión reducida de la clase que gestiona la base de datos, se usó para realizar la implementación y pruebas iniciales sin necesidad de realizar la conexión con una base de datos externa.
- **test:** Paquete que agrupará las clases para realizar las pruebas de funcionamiento y los clientes que se desarrolle. Las clases no son JUnit ya que, aun siendo clases de pruebas, sirven de ejemplo para la implementación de los clientes Java.
- **utils:** En este paquete se disponen de todas las clases que sirven de apoyo a las demás, entre ellas se pueden destacar la clase que sirve de conector con la bases de datos, la que ofrece la funcionalidad para realizar operaciones de cifrado con certificados y la que realiza la conexión con el servicio REST de Composer.
- **WebContent:** Esta carpeta no está dentro del directorio “*src*” del proyecto, está en la raíz del proyecto y es la carpeta que almacena los datos que se envían al servidor web. Dentro se tiene un fichero “*index.html*”, junto con dos librerías JavaScript, que representa el ejemplo de la implementación del cliente web.

Además, el proyecto estará gestionado por Maven de forma que ciertas librerías se integran y actualizan en el mismo de forma transparente. Se han usado dos librerías extras gestionadas por Maven, por un lado “*org.json*” que sirve para el manejo, serialización y parseo de cadenas JSON como objetos, y por otro lado, “*om.github.ipfs*” que sirve de cliente IPFS (se explicará en detalle más adelante).

Tras especificar las partes que forman el esquema para el desarrollo de la solución propuesta queda explicar la implementación de la funcionalidad, en los siguientes apartados se explicará en detalle cómo se realiza la implementación de cada funcionalidad por separado.

4.2.1 Modelo

Como se dijo, existe un paquete llamado “*modelo*” que agrupa las entidades de la plataforma, en la implementación real se disponen de varios modelos ya que se hicieron pruebas con otros sistemas externos (Keyrock) pero para la implementación de la solución propuesta solo es necesario identificar una “*entidad*”.

La entidad principal que gestiona la plataforma son las “*transacciones*” por ello es necesario crear una clase que almacene la información que necesitamos, está será la misma que tienen las

transacciones en la blockchain (el modelo de la BNA) pero teniendo en cuenta que en la del proxy se debe añadir manualmente la ID de la transacción y la marca de tiempo.

Se podría realizar la clase como una clase *JAXB* lo que nos facilitaría muchísimo su gestión al estar directamente representada por un formato XML o JSON, además, Jersey puede hacer el proceso de *marshalling* y /o *unmarshalling* automáticamente al recibir o enviar objetos de esta clase. Aun así, no se opta por esta opción ya que se usará la clase “*JsonObject*” de Java EE 7 para toda la gestión de JSON con objetos.

Tras lo comentado, se puede implementar directamente una clase siguiendo el esquema de *Java Bean*, se dejaría el constructor vacío, todos sus atributos privados y se harían los métodos *GET/SET* que se vean oportunos para respetar la privacidad de los mismos.

A continuación se muestra el encabezado de la clase:

```

1 public class Transaction {
2     private String transactionId;
3     private String cid;
4     private String hash;
5     private String firmaAPI;
6     private String firmaUser;
7     private String timestamp;
8     ....
9     GETs/SETs
10 }
```

Con esta clase la aplicación podrá tener objetos que almacenen los datos que contiene una transacción, además, podrían tener funcionalidad asociada a la misma, aunque en esta implementación no se vio necesaria implementar ninguna.

4.2.2 Utilidades

Este paquete dispone varias clases que tienen por finalidad facilitar algunas tareas que hará la aplicación principal, dentro del mismo existen varias clases pero al igual que pasaba con los modelos unas cuantas de ellas son las que realmente se usan en la implementación final y las son pruebas que se hicieron pero que finalmente no se usarán.

Las tres clases más importantes que tiene este paquete son las explicadas a continuación.

Base de Datos

La clase llamada “*Database.java*” es la que hará de conector con la base de datos y contendrá toda la funcionalidad para realizar consultas o actualizaciones de datos en la misma.

La aplicación principal deberá tener un único objeto de esta clase, el cual al construirse crea una conexión con una determinada base de datos MySQL. La estructura de esa base de datos se muestra en la **figura 19**, como se aprecia es un modelo muy simple ya que solo se requiere guardar los certificados de los usuarios y asociarles un token por sesión.

La clase ofrecerá los siguientes métodos para realizar la funcionalidad requerida:

- **addCertificado(user, cerf, hash):** Este método es una extensión del diseño propuesto para facilitar las pruebas, su finalidad es la de añadir a la base de datos el certificado

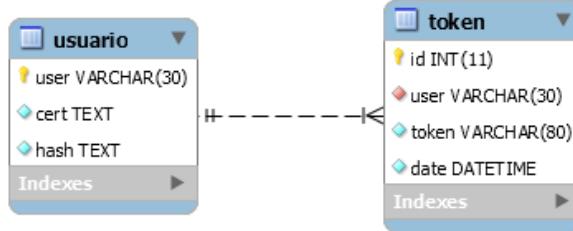


Figura 19: Modelo SQL de la base de datos

público de un determinado usuario (darlo de alta). El método realizará un “*INSERT*” en la base de datos con la información pasada por parámetros y no devolverá nada.

- **addToken(user, token):** Este método realiza otra operación de inserción en la base de datos, añade un token de sesión junto el usuario que lo creó. Este método sirve de apoyo a otro y no devuelve nada.
- **removeToken(token):** Esta función realiza el paso contrario a la anterior, borra de la base de datos un determinado token y no devuelve nada.
- **String getToken(usuario, hash):** Esta es una función muy importante, si finalidad es la de crear y devolver tokens de sesión válidos, para ello recibe como parámetro el usuario y el hash del certificado del mismo y procede a buscarlo en la base de datos, si lo encuentra genera una cadena de texto que servirá de token, llama al método *addToken()* para guardarlo y devuelve la cadena. Si no encuentra al usuario devolverá “0”.
- **boolean checkToken(token):** Este método sirve para comprobar si un token existe en la base de datos (es válido), para ello realiza una simple búsqueda de la cadena indicada en la tabla de tokens. Si lo encuentra devolverá “true” si no “false”.
- **boolean checkUser(user):** Esta función sirve para comprobar si un determinado usuario está dado de alta en la plataforma, esto lo logra haciendo una búsqueda y devolviendo el resultado.
- **String getCertf(user):** Este método sirve para obtener el certificado público de un determinado usuario dado de alta en el sistema, para ello realiza una consulta y devuelve el resultado de la columna que almacena el certificado. Si no encuentra al usuario devuelve nulo.
- **boolean checkCerf(cerf):** Esta función sirve para comprobar si un certificado está almacenado en la base de datos, para ello realiza una consulta sobre la tabla y la columna correspondiente y devuelve “true” si lo encuentra o “false” en caso contrario.

Certificados

La clase almacenada en el fichero “*Certificados.java*” contiene todas las funciones de apoyo para realizar las tareas de cifrado y descifrado RSA haciendo uso tanto de claves privadas como públicas.

Para realizar esto ofrece en primera instancia una función privada para leer ficheros y guardarlos en un array de bytes, paralelamente se ofrecen funciones para crear una clave tanto pública como

privada a partir de un fichero (esos métodos usarán la función privada citada anteriormente) o a partir de una cadena en base64.

Tras disponer de objetos que representen las claves privadas y las públicas podemos usarlos para realizar las tareas de cifrado y descifrado, para ellos se pueden usar los métodos “*encrypt()*” y “*decrypt()*” que ambos están sobrecargados para admitir tanto una clave pública como privada en cualquiera de las operaciones.

API REST de Composer

La clase denominada “*ConektorREST.java*” sirve como conector con el servicio REST de Composer, esta clase codifica el puente entre la plataforma desarrollada y la blockchain. Al igual que la clase de la base de datos, una instancia de esta clase representa una conexión por lo que la aplicación principal deberá tener un único objeto global de esta clase.

La clase ofrece dos métodos que representan toda la funcionalidad que el caso de uso requiere, autenticar y enviar peticiones al servicio REST (consultas y envíos).

El primer método realiza el proceso de autenticación de la plataforma con el servicio REST de Composer, este es necesario ya que se puso el requisito de autenticar el canal por medio de JWT.

Para realizar este proceso el método recibe el JWT que se debe usar y después realiza una conexión HTTP a la dirección del servicio REST con la ruta “*auth/jwt/callback*” pasando como parámetro GET el token JWT, de esta forma la petición devuelve una cabecera en la que se encuentra una cookie con el token de sesión para autenticar todas las comunicaciones.

La petición HTTP se logra gracias a la clase “*HttpURLConnection*” que gestiona una comunicación fácilmente configurable siguiendo el estándar HTTP.

En el siguiente código se muestra cómo se realiza la implementación de esta parte:

```
1 URL url = new URL(direccion + "auth/jwt/callback?token=" + token);
2
3 HttpURLConnection conn = (HttpURLConnection) url.openConnection();
4 conn.setInstanceFollowRedirects(false);
5 conn.setRequestMethod("GET");
6 if (conn.getResponseCode() != HttpURLConnection.HTTP_MOVED_TEMP) {
7     throw new IllegalStateException("Error: " + conn.getResponseCode());
8 }
```

En el código se aprecia la facilidad con la que se puede configurar la conexión, con un método le indicamos que es de tipo GET y con otro más se desactiva la funcionalidad de seguir los redireccionamientos (al autenticar se envía un 302 para redirigirnos a otra dirección) para no perder la cookie.

Cuando se ha realizado con éxito la petición se retorna una respuesta en la que hay un parámetro “*set-cookie*” en la cabecera para indicar que el cliente debe crear una cookie. La cookie que se recibe es la que almacena el token de sesión necesario para autenticar todas las comunicaciones, dado que el objeto que representa la conexión se destruye al terminar el método se debe extraer esa cookie y guardar el valor en un parámetro global del objeto.

Para extraer el valor de la cookie “*access_token*” se debe buscar entre la lista de cookies que contiene la línea “*set-cookie*” de la cabecera recibida, la cual está separada por “;”, alguna con ese nombre y después descomponer el valor (la cookies separa la clave y el valor con “=”) para

obtener el token de sesión, finalmente se tiene que analizar ese valor ya que tiene un formato especial.

El formato de la cookie que devuelve el proceso *callback* del módulo *passport.js* sigue la estructura de “*s:<token>.firmar*” por lo que se debe quitar la primera parte y de eso extraer la cadena antes del punto.

El siguiente código se muestra el proceso en detalle:

```

1 List<String> cookies = conn.getHeaderFields().get("set-cookie");
2 if (cookies != null) {
3     for (String cookie : cookies) {
4         String[] ck = cookie.split(";").split("=");
5         if(ck[0].equals("access_token"))
6             accessToken = java.net.URLDecoder.decode(ck[1], StandardCharsets.UTF_8.name())
7                 .split(":")[1].split("\\.")[0];
8     }
9 }
```

Además del método de autenticación se dispone del método de comunicación con la API REST, este realiza una conexión muy semejante a la del método anterior pero mucho más parametrizada ya que se harán conexiones GET (para consultas) y POST (para envíos) ambas en un formato de cuerpo JSON.

La función recibe como parámetro la acción a realizar, el tipo de conexión y los datos que se quieren enviar. La acción la concatena en la ruta a la que se accederá junto con el token de sesión. Si la petición es de tipo POST se añade al cuerpo de la misma los datos pasados como parámetro. En el siguiente código se muestra cómo se realiza la conexión, se puede apreciar lo semejante que es con el método anterior.

```

1 URL url = new URL(direccion + "api/" + accion + "?access_token=" + accessToken);
2 HttpURLConnection conn = (HttpURLConnection) url.openConnection();
3 conn.setRequestProperty("Content-Type", "application/json");
4 if (post) {
5     conn.setRequestMethod("POST");
6     conn.setDoOutput(true);
7     OutputStream os = conn.getOutputStream();
8     os.write(postData.getBytes());
9     os.flush();
10 } else {
11     conn.setRequestMethod("GET");
12 }
```

Tras realizar correctamente la petición HTTP se lee la respuesta y se devuelve en una cadena como resultado del método. Para leer el resultado del objeto que gestiona la conexión se tiene que obtener el *InputStream* del mismo y meterlo en un buffer de lectura, tras eso se lee línea a línea y se añaden a la cadena que se devolverá. Este proceso se muestra en el siguiente código:

```

1 BufferedReader br = new BufferedReader(new InputStreamReader(conn.getInputStream()));
2
3 StringBuffer response = new StringBuffer();
4 String readLine = null;
5 while ((readLine = br.readLine()) != null) {
6     response.append(readLine);
7 }
8 br.close();
9 conn.disconnect();
10 return response.toString();
```

Con lo explicado se dispondrá finalmente de tres clases de utilidades que serán utilizadas multitud de veces en la implementación final.

4.2.3 Servicio REST

El paquete “rest” es el que contiene todos los servicios REST que Jersey mapeará automáticamente dentro de un *servlet* y les agregará la funcionalidad extra que ofrece por medio de anotaciones. Jersey sabe que este paquete es el que debe analizar en busca de servicios porque se le indicó en el fichero de configuración de Apache Tomcat (*web.xml*) la ruta del paquete, a continuación se muestra un fragmento representativo del fichero.

```

1 <servlet>
2 ...
3   <init-param>
4     <param-name>com.sun.jersey.config.property.packages</param-name>
5     <param-value>rest</param-value>
6   </init-param>
7 </servlet>
8 <servlet-mapping>
9   <servlet-name>jersey-servlet</servlet-name>
10  <url-pattern>/iot-bna/*</url-pattern>
11 </servlet-mapping>
```

En la implementación realizará solo se dispone de una clase dentro del paquete la cual se denomina “*IoTRestApp.java*”, esta clase agrupa toda la funcionalidad que ofrece la plataforma. Jersey gestionará de forma transparente el ciclo de vida de la clase, creando objetos cuando vea necesario.

En el constructor de la clase se realizan las tareas de inicialización, primero construye el objeto de conexión con la API de Composer y se realiza una invocación al método para autenticarla, tras eso se inicializa el objeto de la librería de cifrado Alice y se construye el cliente IPFS.

Anteriormente dijimos que la conexión con el nodo IPFS local era a través de una API REST, pero en la implementación se ha simplificado esta tarea haciendo uso de una librería gestionada por Maven (“github.com/ipfs/java-ipfs-http-client”) la cual únicamente requiere construir un objeto de la clase “IPFS” y pasarle como parámetro la ruta del servicio REST del nodo. Tras crear el objeto correctamente se dispone del método “*ipfs.add(file)*” el cual añade un fichero en el nodo y devuelve el CID asignado.

Finalmente, la clase del servicio tiene una serie de parámetros que representan variables globales, entre ellas está el token JWT que se utilizará para la autenticación con Composer y la clave privada del certificado de la aplicación que servirá para firmar las transacciones.

En los siguientes apartados se explicará la realización de los métodos que ofrece la plataforma.

Certificados

En la implementación de la solución propuesta se han creado dos métodos para la gestión de certificados.

Uno de los métodos sirve para añadir certificados a la base de datos, como ya se dijo, es una funcionalidad extra que en un escenario real posiblemente no se hubiera hecho pero para simplificar las pruebas se ha creado. El método es accesible mediante la URL de la aplicación seguido de “/cert”, este requiere que los datos se envíen en formato JSON siguiendo la siguiente estructura:

```

1 {
2     "cert" : <certificadoPEM>,
3     "user" : <nombreUsuario>
4 }
```

Como se ve, se le debe pasar un campo con el certificado en Base64 (estándar PEM) y otro campo con un nombre de usuario que asociará al certificado y usará como clave primaria, si se quisiese evitar este campo por motivos de privacidad se podría usar el hash del certificado como sustituto.

Tras recibir la petición el método inserta en la base de datos el certificado asociado al usuario siempre que no exista uno ya.

Por otro lado, el otro método sirve para realizar la petición de un certificado público de un determinado usuario, este es accesible mediante la ruta “/certificado/{usuario}” donde el último parámetro representa el nombre de usuario del que se quiere obtener el certificado.

Esta función es útil para los ejemplos en los que los clientes de la plataforma no disponen de los certificados de los otros usuarios que necesitarán para descifrar lo que ellos cifren usando su clave privada o para los casos en los que un cliente quiera cifrar con la clave pública de otro para que solo ese lo pueda descifrar. En general es una función para poder dar públicamente los certificados de la plataforma.

Proceso de Autenticación

Para realizar todo el proceso de autenticación de los clientes se han realizado varios métodos para cubrir la funcionalidad requerida.

Inicialmente se implementa una función de apoyo para el resto de métodos de la aplicación que requieran estar autenticados, esta función privada denominada “isValidToken(token)” recibe como parámetro un token de sesión que deberá comprobar si es válido, para ello lo busca en la base de datos y si lo encuentra retorna el valor “true”, en caso contrario “false”.

Para usarlo en los métodos que se quiera requerir autenticación se deberá simplemente llamar al comienzo del código a esta función pasándole el token recibido y en caso de ser válido seguir la ejecución, si el token no es válido se retorna directamente un mensaje de error.

El siguiente método es el más importante de todos, es al que se debe acceder para realizar la autenticación de un cliente, para invocarlo se debe acceder a la ruta “/auth” y enviar en el cuerpo un mensaje JSON con la siguiente estructura:

```

1 {
2     "cert" : <certificadoPEM>,
3     "user" : <nombreUsuario>
4 }
```

Tras eso se buscará en la base de datos el certificado de ese usuario y se comparará para asegurar que está dado de alta, como se explicó en el diseño, es posible sustituir este paso por una comprobación de la entidad certificadora que firma el certificado recibido, si el certificado lo ha expedido una CA dada de alta en la base de datos se considera el certificado directamente válido.

Si el certificado es válido se procede a la creación de un token llamando al método “getToken(user, hashCert)” del objeto que sirve de conector con la base de datos, después se cifra el token con el

certificado público del cliente (usando la utilidad citada anteriormente) y se le envía una respuesta JSON con el formato:

```

1 {
2     "token" : <tokenCifrado>
3 }
```

Si el cliente dispone de la clave privada del certificado podrá descifrar el token que representa la sesión que acaba de crear, ese token lo tendrá que usar en todos los métodos que requieran autenticación.

Para probar si se logra la autenticación correctamente se implementa un sencillo método al cual se accede desde la ruta “/testAuth” y se le pasa el token como un parámetro GET, el método utilizará la función “isValidToken()” para comprobar si es válido y devolverá el resultado.

Finalmente, se realiza un último método al cual se accede con la ruta “/logout” y que sirve para cerrar una sesión del token pasado como parámetro GET. El método únicamente borra de la base de datos la entrada que contiene el token recibido, de esta forma cualquier otra petición que use ese token dejará de ser válida.

Consultar Transacciones

En la plataforma se proporcionan diversos métodos para realizar consultas, todos ellos requieren que el cliente les envíe un token válido para asegurar que están autenticados.

El primer método de consulta se accede mediante la ruta “/transactions” y recibe el token como un parámetro GET, como resultado devuelve la lista de transacciones que están almacenadas en la plataforma en formato JSON según la siguiente estructura:

```

1 [
2     { "transactionId" : <IDtransaccion>,
3      "timestamp" : <marcaTiempo>,
4      "cid" : <CIDenIPFS>,
5      "hash" : <resumenDatos>,
6      "firmaUser" : <firmaUsuarioDatos>,
7      "firmaAPI" : <firmaProxyDatos>
8    },
9    ....
10 ]
```

Para conseguir eso se hace uso de las clases “*JSONArray*” y “*JSONObject*” que facilitan el proceso de iteración sobre el JSON recibido de la API de Composer, además, en cada iteración se construye un objeto “Transaction”, se actualizan sus parámetros y se añade a una lista.

Tras iterar sobre todas las transacciones se serializa la lista en formato JSON y se devuelve al cliente. Este proceso se ve necesario ya que, en vez de enviar directamente el JSON recibido al cliente, se prefiere construir uno con la estructura y datos que se vean oportunos, aunque en la práctica tiene un formato similar al que usa la API de Composer.

Además del método para obtener todas las transacciones se dispone de otro para obtener solo una indicando en la ruta la id de la misma (“/transactions/{id}”). El funcionamiento de este método es el mismo que el anteriormente explicado.

Una cuestión interesante, que quedaba fuera del ámbito de este documento, es la posibilidad de aplicar técnicas RBAC (“Role-based access control” - “Control de acceso basado en rol”) para

definir roles y, aprovechando el hecho de que todos los clientes están debidamente autenticados, limitar el uso de los métodos de consultas y listados a un determinado grupo de usuarios.

Por último, quedan tres métodos relacionados con los datos que se almacenan, el primero de ellos tiene también un funcionamiento semejante a los anteriores, sirve para obtener una determinada transacción pero usando el hash de los datos como identificador. Para acceder a esta función se debe enviar la petición a la ruta “`/find/{hash}`”, después el método pedirá todas las transacciones persistidas en la blockchain y buscará una que coincida con el hash indicado, tras eso la devolverá en JSON con la misma estructura ya comentada.

El segundo método es idéntico al anterior pero recibe directamente los datos, tras eso calcula el hash y realiza el mismo proceso indicado anteriormente. Se accede a la función enviando una petición POST en la ruta “`/find/`” en formato JSON con los campos “`token`” y “`data`”

El último método que queda tiene un propósito más simple, únicamente devuelve los datos almacenados en el IPFS al pasarle el CID como parámetro en la ruta, para ello se debe realizar una petición a la dirección “`/ipfs/{cid}`”. La función usará la librería IPFS que se explicó al principio de esta sección, para ello llamará al método “`ipfs.cat(cid)`” al que le pasa el CID en formato *Multihash* y este devolverá un array de bytes con los datos.

Como conclusión, el uso conjunto de estos métodos sirve para lograr la verificación de la producción de un determinado dato o documento en un momento concreto, de esta forma se usaría el hash de los datos para buscar la transacción y de ella se podría obtener la fecha de creación y el CID con los datos cifrados para poder realizar el proceso de validación.

Crear Transacción

Finalmente queda por explicar el último método disponible, este es el encargado del proceso para persistir los datos y crear las transacciones en la blockchain.

El método es accesible de dos maneras, ambas realizando una petición a la ruta “`/transaction`”, si se desea enviar los datos ya cifrados (escenario típico) se debe realizar una petición de tipo POST, por el contrario, si se envían los datos sin cifrar y se quiere que el proxy los cifre usando una política interna basada en la identidad del cliente (no se ha implementado, la clave está *hardcoded*) se debe realizar una petición de tipo PUT.

El formato del contenido que se envía en la petición sigue siendo JSON con la siguiente estructura:

```
1 {
2     "token" : <tokenSesion>,
3     "firma" : <firmaHash>,
4     "data"   : <datos>
5 }
```

Tras eso, ambos métodos llaman a la función “`dataPush()`” que se encarga de todo el proceso de envío. Lo primero que hace es extraer los parámetros recibidos en la petición y guardarlos en variables, después calcula el hash de los datos recibidos y genera la firma de ellos con la clave privada del proxy (usando la utilidad de cifrado) y por último, envía los datos al IPFS usando el método “`ipfs.add(file)`” de la librería y obtiene el CID.

Con todos los parámetros calculados se procede al envío de la petición POST a la API de Composer para que cree una transacción, el mensaje está en JSON con la estructura:

```

1 {
2   "$class"      : "iot.IoTTransaction",
3   "cid"         : <cidIPFS>,
4   "hash"        : <hashDatos>,
5   "firmaAPI"    : <firmaHashAPI>,
6   "firmaUser"   : <firmaHashUser>
7 }

```

Al realizar la petición con éxito Composer envía una respuesta con todos los datos de la transacción, la función construye un objeto “Transaction” con los datos recibidos y lo envía como respuesta (serializada en JSON) al cliente.

4.3 Implementación de los Clientes

En la implementación del caso de uso de la solución propuesta se han desarrollado tres softwares clientes para servir de ejemplo en la utilización del sistema. Dos de los ejemplos están basados en los escenarios de uso por dispositivos que se diseñaron en la sección anterior, sirviendo como *PoC* de un uso real que podría tener la plataforma.

En los siguientes apartados se explicará la implementación de cada ejemplo.

4.3.1 Ejemplo Java

El primer ejemplo de aplicación cliente no está basado en ningún escenario de uso en concreto, más bien, se puede tratar el ejemplo como una clase de pruebas que sirve para probar la funcionalidad de la plataforma y de ejemplo de uso.

El lenguaje usado para la implementación del código es Java para que la clase pueda formar parte del proyecto del proxy, este se escribe en el fichero “*ClienteProxyTest.java*” del paquete “*test*” y está formado por una serie de métodos de pruebas envueltos en un método estático.

Los métodos realizan las tareas de autenticación, listado de transacciones, prueba de token de sesión, envío de datos, obtención de datos en el IPFS y eliminación de tokens. La clase se ha usado a lo largo del desarrollo para probar y verificar las modificaciones que se hacían en la implementación, además, el código realizado ha servido de base para la implementación del cliente IoT que se explicará en el apartado final de esta sección.

Las credenciales que usa la clase para el proceso de autenticación las tiene almacenadas en variables globales, aunque está implementado el código necesario para realizar la lectura desde ficheros PEM.

4.3.2 Ejemplo Web

En este ejemplo desarrolla una *prueba de concepto* basada en el escenario de uso de dispositivos personales, en este ejemplo los clientes son usuarios normales que desean persistir documentos en la plataforma por lo que el acceso debe ser fácil y la interfaz simple.

Para realizar la aplicación de ejemplo se implementa una aplicación Web escrita en HTML y JavaScript. La aplicación hace uso de las siguientes librerías para facilitar el desarrollo:

- **JQuery:** Es una biblioteca multiplataforma de JavaScript que permite simplificar la manera de interactuar con la web, manejar eventos y agrega interacción con otras páginas web usando la técnica AJAX.
- **Bootstrap:** Es un framework CSS y JavaScript diseñado para la creación de interfaces simples y responsive. Además, ofrece un amplio abanico de herramientas y funciones que facilitan la creación de páginas de cualquier propósito.
- **FontAwesome:** Es un framework de iconos vectoriales y estilos CSS, esto permite hacer uso de una gran cantidad de recursos gráficos basados en fuentes por lo que se reduce el uso del ancho de banda.
- **JSEncrypt:** Es una librería JavaScript para realizar las tareas de encriptación, desencriptación y generación de claves en OpenSSL RSA.
- **JS-sha256:** Es una librería simple de JavaScript para calcular el resumen de unos datos haciendo uso de la función SHA256.

El código de la página desarrollada se encuentra en el fichero “*index.html*” dentro del directorio “*WebContent*” del proyecto Java Web. En la **figura 20** se muestra la apariencia visual del prototipo desarrollado.

Subir documento en la plataforma

Figura 20: Ejemplo de aplicación Web

La aplicación dispone de una única página con un formulario compuesto de tres campos para seleccionar archivos, el primero de ellos es para seleccionar el fichero con la clave privada del cliente, el segundo es para seleccionar el certificado público y el último campo es para seleccionar el documento que se quiere persistir en la blockchain.

En la parte inferior del formulario hay un botón que sirve para realizar el proceso de envío, dando al finalizar un mensaje con el identificador de la transacción y el CID del fichero.

Para realizar el proceso de envío del documento la aplicación realiza varios pasos, hay que recordar que el código es JavaScript por lo que se ejecuta de forma local en el compilador del explorador y no se envía nada fuera del mismo (la clave privada solo se usa localmente).

Al presionar el botón para enviar el documento la aplicación procede al proceso de autenticación

con el proxy, para ello carga en la memoria local el certificado público del cliente y prepara un envío AJAX con ese certificado hacia la plataforma para que le devuelva un token de sesión.

Tras recibir la respuesta carga también en memoria local la clave privada del usuario y haciendo uso de la librería *JSEncrypt* desencripta el token recibido para poder usarlo. Si logra descifrar el token significa que el proceso de autenticación se ha completado con éxito por lo que puede realizar el envío en sí, para ello lee el contenido del documento, lo guarda en una variable, calcula el hash y lo firma con la clave privada. Las tres variables recién calculadas son empaquetadas en un mensaje JSON y se envían por AJAX al proxy para que cree la transacción.

Se debe tener en cuenta que, por comodidad para verificar el caso de uso, el documento que se envía no se ha cifrado para que se persista en el IPFS en texto claro. Para cifrar el dato se podría hacer de varias maneras, una de ellas sería la de realizar la misma operación de cifrado con la clave privada que se le hizo al hash para hacer la firma, otra sería la de requerir al cliente algún tipo de clave local para realizar un cifrado simétrico, y otra más sería que el cliente suba directamente el documento cifrado con las técnicas que vea oportunas.

Finalmente se recibe una respuesta del proxy en JSON, que en Javascript es directamente utilizable como un objeto, de la cual se extrae dos de los campos que tiene disponible, el identificador de la transacción creada y el CID de los datos guardados en el IPFS, para mostrar un mensaje en pantalla con esa información.

4.3.3 Ejemplo IoT

Para la realización del ejemplo de aplicación IoT, basada en el escenario de uso de dispositivos IoT, se crea una aplicación Java por la ventaja que ofrece la máquina virtual de Java al ser multiplataforma, de esa forma se puede ejecutar directamente en una Raspberry la aplicación sin ningún tipo de inconveniente ni configuración especial. Además, se podría realizar una ejecución semejante en cualquier dispositivo compatible con Java.

El software de ejemplo está disponible en la clase “*ClienteIoTApp.java*” del paquete “*test*”. La clase tiene un método estático principal que sirve de punto de ejecución inicial, en él primeramente se realiza la lectura de los ficheros que guardan el certificado público y la clave privada del mismo, esto se hace para simular la lectura de los certificados que traen de fábrica estos dispositivos.

Al disponer del certificado se puede realizar el proceso de autenticación de la aplicación, para ello realiza una conexión a la ruta de autenticación junto con el certificado público recién leído. La petición retorna un token de sesión cifrado con el certificado público por lo que para usarlo es necesario descifrarlo haciendo uso de la clave privada y la utilidad de cifrado RSA desarrollada.

Si se consigue descifrar el token correctamente se pueda dar por completado con éxito el proceso de autenticación, disponiendo de un identificador de sesión que se deberá utilizar en todas las comunicaciones futuras.

Con la sesión autenticada disponible la aplicación puede comenzar el envío de datos a la plataforma. En la implementación se ha realizado un ejemplo sencillo de envío de información de los sensores, para ello se realiza un bucle que itera unas pocas veces y que deja un retardo de unos segundos entre iteración.

Dentro del bucle se simula una lectura de los tres sensores (temperatura, presión y nivel) generando los valores con funciones seno que dependen del índice del bucle. Después, esas lecturas junto con un identificador del sensor se empaquetan en un mensaje JSON que representará

la información a persistir (típicamente estos dispositivos producen la información así). Tras eso, se calcula el resumen de ese mensaje y se genera la firma del mismo (con la clave privada).

Finalmente, se juntan los datos, el hash, la firma y el token de sesión en un JSON que se envía al proxy para que realice el proceso para persistir la información en la blockchain.

A continuación se muestra un ejemplo de mensaje JSON generado por la aplicación:

```
1 {  
2     "token" : "55eb9d5a-da6d-4956-bd30-ce3491024672",  
3     "firma" : "13a81859392cce791ccc4bfe1549ed437e3f8873a5c11ca32cfb87b9ae8df98",  
4     "data" : "{\"sensor\": \"sensor-01\", \"temperatura\": \"53.48\",  
5                 \"presion\": \"10.17\", \"nivel\": \"206.97\"}"  
6 }
```

Cuando el bucle termina de realizar todas las iteraciones se considera que la ejecución del ejemplo ya ha terminado por lo que se envía una última petición a la plataforma para realizar un proceso de *logout* en el que se debe invalidar el token de sesión que se estaba usando.

Para hacer uso de la aplicación en un dispositivo Raspberry se debe compilar el código en un paquete JAR, enviarlo al dispositivo y ejecutarlo haciendo uso de Java, en el sexto anexo se muestra el proceso en detalle.

4.4 Resumen final de la implementación realizada

A modo de resumen del trabajo realizado, inicialmente se propusieron unos objetivos que fueron tratados en el diseño de las soluciones, de esos diseños se opta por realizar la implementación de uno de ellos, de forma simplificada, ya que se desea realizar un ejemplo de plataforma como prueba de concepto del sistema.

La implementación realizada cumple con las especificaciones planteadas, sirve de punto de interconexión entre una blockchain y los clientes, añade mecanismos para aumentar la seguridad como hacer uso de un canal cifrado, proporciona herramientas para realizar un proceso de autenticación de las partes involucradas, usa propiedades criptográficas para certificar los datos manejados y añadir capas de privacidad, hace uso de sistemas distribuidos para persistir los datos y ofrece un acceso ligero para los clientes menos potentes. Por todo esto se dan por cumplidos los requisitos descritos en la [sección 3.1](#).

Como se ha dicho, el sistema se desarrolla como una aplicación Java Web, en ella se realiza un servicio REST que mapea de forma autodescriptiva distintas rutas a los procedimientos que ofrecen cada funcionalidad. Toda esta comunicación se realiza mediante el estándar HTTP (y para más seguridad HTTPS/TLS) y en todos los mensajes el formato del cuerpo es JSON.

Los distintos métodos disponibles sirven para realizar las especificaciones del sistema, autenticación del cliente, consultar listados de transacciones, transacciones concretas, documentos en el IPFS, buscar una transacción por hash y finalmente realizar un envío de datos a la blockchain.

En resumen, los métodos que ofrece el Blockchain Security Handler son los siguientes:

- **isValidToken()**: Función privada de apoyo para comprobar si un token es válido, si está en la base de datos.
- **certificadoAdd()**: Se invoca con un mensaje POST en la ruta “/cert”, se pasa un certificado y un identificador y sirve para dar de alta un determinado certificado público en la plataforma.
- **getCerft()**: Se invoca con un mensaje GET en la ruta “/certificado/{user}”, sirve para devolver el certificado público de un determinado usuario.
- **testAuth()**: Se invoca con un mensaje GET en la ruta “/testAuth”, sirve para comprobar si el token pasado como parámetro GET es válido o no.
- **autenticar()**: Se invoca con un mensaje POST en la ruta “/auth”, se pasa un certificado y un identificador y sirve para realizar el proceso de autenticación, si el certificado es válido se genera un token y se envía cifrado con el certificado al cliente.
- **logout()**: Se invoca con un mensaje GET en la ruta “/logout”, sirve para borrar el token pasado como parámetro GET de la base de datos, se usa como método para cerrar una determinada sesión.
- **getTransactions()**: Se invoca con un mensaje GET en la ruta “/transactions”, sirve para obtener un listado de las transacciones del sistema.
- **getTransaction()**: Se invoca con un mensaje GET en la ruta “/transaction/{id}”, sirve para obtener los detalles de la transacción con el identificador indicado en la ruta.
- **transactionPostData()**: Se invoca con un mensaje PUT en la ruta “/transaction”, sirve para realizar el envío de los datos (crear una transacción) pasando los datos sin cifrar.

- **transactionPutData()**: Se invoca con un mensaje POST en la ruta “*/transaction*”, sirve para realizar el envío de los datos (crear una transacción) pasando los datos ya cifrados.
- **dataPush()**: Función privada de apoyo para realizar la tarea de creación de una transacción, recibe los datos con la firma del cliente, calcula el hash, su firma y envía un mensaje con todos esos parámetros a la blockchain.
- **getTransactionByHash()**: Se invoca con un mensaje GET en la ruta “*/find/{hash}*”, sirve para obtener los detalles de una determinada transacción identificada por el hash de los datos.
- **getTransactionByData()**: Se invoca con un mensaje POST en la ruta “*/find/*”, se pasan los datos y sirve para obtener los detalles de una determinada transacción identificada por el hash de los datos.
- **getIPFS()**: Se invoca con un mensaje GET en la ruta “*/ipfs/{cid}*”, sirve para obtener los datos del nodo IPFS de un determinado CID.

Todos estos métodos son accesibles con facilidad mediante clientes basados en el estándar HTTP.

Además, se han realizado tres ejemplos para probar la funcionalidad de la plataforma, el primero de ellos es una mera clase de pruebas en Java, el segundo representa el escenario de uso de un usuario de escritorio que accede vía web al sistema para persistir documentos, y el tercer ejemplo muestra un uso desde un dispositivo IoT con sensores que envía constantemente los datos para dejar constancia de la información que lee.

Gracias a los ejemplos de clientes y a la propia implementación del servicio proxy es posible realizar una prueba del sistema propuesto por completo (teniendo en cuenta las limitaciones del diseño simplificado), de esta forma es posible probar varios escenarios con distintos tipos de clientes, cada uno de ellos con unas necesidades y requisitos concretos. Con las implementaciones y las pruebas es posible demostrar de forma exitosa el funcionamiento y la utilidad de la solución tratada en este documento, dando por cubierto los **objetivos 5.1, 5.2 y 5.3**.

Para terminar, todo el trabajo realizado para concluir este documento será de carácter público, tanto el software desarrollado para el caso de uso como el código LATEX de la memoria serán publicados en el siguiente GitHub para que puedan servir de inspiración y ayuda a cualquiera que quiera indagar en este interesante campo.

<https://github.com/alb1183/TFG>

5 Conclusiones y Vías futuras

Tal y como se ha ido comentando a lo largo de este documento, las tecnologías que se han usado para desarrollarlo poseen en la actualidad una creciente popularidad, están en constante evolución y poco a poco se aplican en proyectos de gran envergadura por parte de empresas reputadas del sector. Aun así, siguen siendo tecnologías bastante nuevas y les queda mucho por delante, esto hace que sea difícil iniciarse en ellas, pero por el contrario, es beneficioso para toda persona o entidad que se adentre en este campo para investigar ya que podrá tener una participación destacable y ayudará a avanzar estas tecnologías novedosas.

El *granito de arena* que se ha aportado con la solución propuesta ha sido el de agrupar un conjunto de ideas novedosas para realizar el diseño e implementación de una plataforma aparentemente simple, pero con una finalidad muy interesante y que en un futuro será un ejemplo de sistema imprescindible. En verdad, es posible percibir que la corriente de investigación en el mundo IoT y Blockchain tiende hacia temas de seguridad, privacidad, trazabilidad y mecanismos de autenticación y certificación, por lo que se considera que el trabajo realizado será una aportación útil dentro de este campo.

Aun con lo dicho, la implementación ha sido simplificada y el diseño más interesante no se ha podido abarcar en el caso de uso por distintas limitaciones, por ello surge directamente una posible vía futura muy necesaria que consistiría en explorar en detalle el diseño de autenticación federada y realizar una implementación para validar su funcionamiento.

Otro punto interesante que no se ha tratado han sido los mecanismos de autenticación, en el caso del diseño se optó directamente por el uso de certificados, pero en el contexto IoT existen otros mecanismos de actualidad con mucho futuro como los basados en identificadores descentralizados[18], por lo que se podría plantear como vía futura el análisis, diseño e implementación de un soporte para realizar este tipo de autenticación en los clientes IoT.

Además, relacionado con lo anterior, los mecanismos de cifrados de los datos quedaron totalmente fuera de nuestro interés, para un caso de uso concreto se propuso el cifrado asimétrico, por lo que sería de útil analizar otras alternativas posibles. Dentro de las opciones existen unas recientes, muy interesantes en el ámbito distribuido del IoT, denominadas técnicas de cifrado basadas en políticas de atributos, por ejemplo, CP-ABE (*Ciphertext-Policy Attribute-Based Encryption*)[26][27].

Las técnicas de cifrado basadas en políticas de atributos consisten en realizar el cifrado en función de unos determinados parámetros o atributos que cumpla el dispositivo, al cumplirlos dispondrá del material criptográfico asociado a cada uno de ellos y podrá usarlo en un proceso de cifrado que tenga en cuenta los atributos que se vean convenientes.

De esta forma se cifrarían los datos en base a unos atributos concretos, y todos los dispositivos que los cumpliesen podrían hacer uso de sus claves asociadas a los mismo para descifrar los datos. Se conseguiría un mecanismo para limitar el ámbito de privacidad de los datos de una manera flexible y segura, por ello los clientes podrían compartir datos sensibles en el blockchain de forma segura solo con aquellos usuarios y/o entidades que satisfagan una serie de atributos de identidad.

Respecto a los ejemplos implementados también es posible realizar mejoras en la funcionalidad. Por un lado, en el caso del ejemplo IoT se podría directamente desarrollar un cliente para un

dispositivo real que capture los datos de unos sensores y forme parte de una autentica red IoT. El ejemplo serviría para demostrar su uso en un escenario en producción real.

Por otro lado, en el ejemplo de aplicación web podría mejorarse mucho más para que tenga un enfoque distinto, mantendría el caso de uso de usuarios que desean persistir documentos pero se propondría como un complemento del navegador web de tal forma que los usuarios simplemente tendrían que hacer uso de un pequeño ícono en el navegador web donde se pediría el documento y la credenciales se gestionarían usando el banco de certificados del sistema operativo.

Como se ve, se pueden desarrollar otro tipo de aplicaciones que hagan uso de la plataforma para distintos escenarios, al proporcionar una interfaz REST se facilita enormemente la tarea de crear aplicaciones a los desarrolladores que quieran usar el sistema.

Finamente, se puede concluir que se han logrado todos los objetivos planteados. Gracias a la investigación realizada se ha aprendido mucho sobre Hyperledger, sobre temas de seguridad en entornos IoT y se ha realizado la implementación de una plataforma de ejemplo verificable en un caso de uso práctico. Paralelamente se ha realizado un estado del arte de las tecnologías actuales para contextualizar el diseño de la solución.

Como trabajo futuro se tiene todo aquello que quedó fuera de estudio o realización, lo más inmediato es la implementación del diseño federado para la realización de la autenticación haciendo uso de eIDAS junto FiWare. Otro tema que inicialmente se trató fue el de realizar el proceso de autenticación por medio de contratos inteligentes, pero este finalmente no se realizó por preferir el esquema actual, se podría estudiar cómo llevar esto a la práctica.

References

- [1] Mennan Selimi, Aniruddh Rao, Anwaar Ali, Leandro Navarro, and Arjuna Sathiaseelan. Towards blockchain-enabled wireless mesh networks. 04 2018.
- [2] FiWare. Connecting IdM Keyrock to a eIDAS Node. <https://fiware-idm.readthedocs.io/en/latest/eidas/architecture/index.html>, 2019.
- [3] Confederación Española de Organizaciones Empresariales. Plan Digital 2020: la digitalización de la sociedad española. http://contenidos.ceoe.es/CEOE/var/pool/pdf/publications_docs-file-334-plan-digital-2020-la-digitalizacion-de-la-sociedad-espanola.pdf, 2016.
- [4] Juri Mattila and Timo Seppälä. Blockchains as a Path to a Network of Systems - An Emerging New Trend of the Digital Platforms in Industry and Society. ETLA Reports 45, The Research Institute of the Finnish Economy, August 2015.
- [5] H. Cai, B. Xu, L. Jiang, and A. V. Vasilakos. Iot-based big data storage systems in cloud computing: Perspectives and challenges. *IEEE Internet of Things Journal*, 4(1):75–87, Feb 2017.
- [6] Martin Strohbach, Holger Ziekow, Vangelis Gazis, and Navot Akiva. *Towards a Big Data Analytics Framework for IoT and Smart City Applications*, pages 257–282. Springer International Publishing, Cham, 2015.
- [7] U. S. Shanthamallu, A. Spanias, C. Tepedelenlioglu, and M. Stanley. A brief survey of machine learning methods and their sensor and iot applications. In *2017 8th International Conference on Information, Intelligence, Systems Applications (IISA)*, pages 1–8, Aug 2017.
- [8] J. Cañedo and A. Skjellum. Using machine learning to secure iot systems. In *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pages 219–222, Dec 2016.
- [9] Muhammad Salek Ali, Koustabh Dolui, and Fabio Antonelli. Iot data privacy via blockchains and ipfs. 10 2017.
- [10] Blockchain Development on Hyperledger Fabric using Composer. <https://www.udemy.com/hyperledger/>, 2018.
- [11] Martijn Bastiaan. Preventing the 51%-attack: a stochastic analysis of two phase proof of work in bitcoin. 2015.
- [12] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. <https://bitcoin.org/bitcoin.pdf>, 2008.
- [13] Walid Al-Saqaf and Nicolas Seidler. Blockchain technology for social impact: opportunities and challenges ahead. *Journal of Cyber Policy*, 2(3):338–354, 2017.
- [14] Blockchain Technology Projects - Hyperledger. <https://www.hyperledger.org/projects>, 2019.
- [15] Hyperledger. Una breve introducción. <https://www.versia.com/blog/hyperledger-una-breve-introduccion.html>, 2018.

- [16] Wikipedia contributors. Hyperledger - Wikipedia. <https://en.wikipedia.org/w/index.php?title=Hyperledger&oldid=904800409>, 2019.
- [17] Hyperledger: la Blockchain privada que todos tenemos que conocer. <https://www.eleconomista.es/economia/noticias/8899454/01/18/Hyperledger-la-Blockchain-privada-que-todos-tenemos-que-conocer.html>, 2018.
- [18] Yki Kortesniemi, Dmitrij Lagutin, Tommi Elo, and Nikos Fotiou. Improving the privacy of iot with decentralised identifiers (dids). *Journal of Computer Networks and Communications*, 2019:1–10, 03 2019.
- [19] Ahsan Manzoor, Madhusanka Liyanage, An Braeken, Salil S. Kanhere, and Mika Ylianttila. Blockchain based proxy re-encryption scheme for secure iot data sharing. 11 2018.
- [20] Syed Saud Hasan, NAZATUL SULTAN, and Ferdous Ahmed Barbhuiya. Cloud data provenance using ipfs and blockchain technology. pages 5–12, 07 2019.
- [21] Oleksii Konashevych. The use of the blockchain technology for public key infrastructure of eidas. 2019.
- [22] Michael Maliappis, Kostis Gerakos, Constantina Costopoulou, and Maria Ntaliani. Authenticated academic services through eidas. *International Journal of Electronic Governance*, 11:1, 01 2019.
- [23] Jorge Bernal Bernabe, José Hernández-Ramos, and Antonio Skarmeta. Holistic privacy-preserving identity management system for the internet of things. *Mobile Information Systems*, 2017:20 pages, 08 2017.
- [24] Khaled Salah and Minhaj Ahmad Khan. Iot security: Review, blockchain solutions, and open challenges. *Future Generation Computer Systems*, 11 2017.
- [25] Xiaoqi Li, Peng Jiang, Ting Chen, Xiapu Luo, and Qiaoyan Wen. A survey on the security of blockchain systems. *Future Generation Computer Systems*, 08 2017.
- [26] José L Hernández-Ramos, Salvador Pérez, Christine Hennebert, Jorge Bernal Bernabé, Benoit Denis, Alexandre Macabies, and Antonio F Skarmeta. Protecting personal data in iot platform scenarios through encryption-based selective disclosure. *Computer Communications*, 130:20–37, 2018.
- [27] S Roy and M Chuah. Secure data retrieval based on ciphertext policy attribute-based encryption (cp-abe) system for the dtns. *Lehigh CSE Tech. Rep.*, 2009.

Anexos

Anexo I: Despliegue de Hyperledger

Hyperledger tiene distintas maneras de ser desplegado dependiendo de las necesidades que tengamos. Por un lado, se puede hacer un despliegue dedicado sobre máquinas físicas (enfoque nativo), instalando todas las dependencias de forma manual en cada una, esto es útil para escenarios en producción ya que cada equipo se encarga de ejecutar de forma independiente y segura cada módulo de Hyperledger. Por otro lado, es posible hacer un despliegue más simple (enfoque virtual) haciendo uso de contenedores Docker para la ejecución de los módulos sobre una máquina virtual Vagrant (VirtualBox), este despliegue es muy interesante para nosotros ya que es muy fácil de realizar, permite hacer pruebas de concepto fácilmente, no requiere que instalemos nada (ya que el equipo de Hyperledger ofrece las maquinas preconfiguradas) y además es portable.

Instalación Vagrant

Para la instalación de Vagrant se requiere previamente instalar un hipervisor que gestione la máquina virtual, en nuestro caso usaremos VirtualBox por ser una alternativa gratuita.

<https://www.virtualbox.org/wiki/Downloads>

Tras su instalación ya es posible instalar Vagrant, para ello visitamos la página de descargar y obtenemos el instalador.

<https://www.vagrantup.com/downloads.html>

Creación de la máquina Vagrant

Una vez tenemos Vagrant y VirtualBox instalados ya podemos comenzar el despliegue de Hyperledger, para ellos debemos descargarnos del repositorio oficial la “imagen” de Vagrant, en nuestro caso usaremos la versión modificada del curso de Udemy con el que me he formado[10] ya que dispone de utilidades que facilitaran el despliegue.

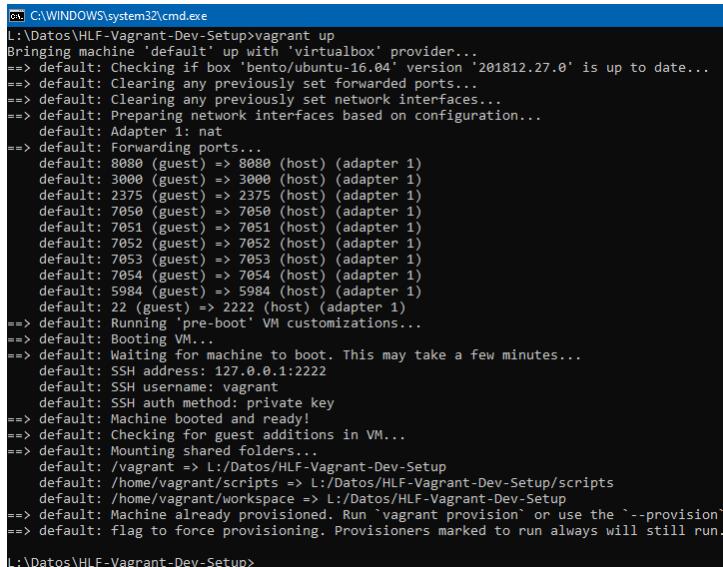
<https://github.com/acloudfan/HLF-Vagrant-Dev-Setup>

Dentro del directorio del repositorio se debe ejecutar el comando de arranque de la máquina Vagrant, dado que será la primera vez que la encendemos tardará unos minutos en estar operativa ya que deberá descargar todos los requisitos.

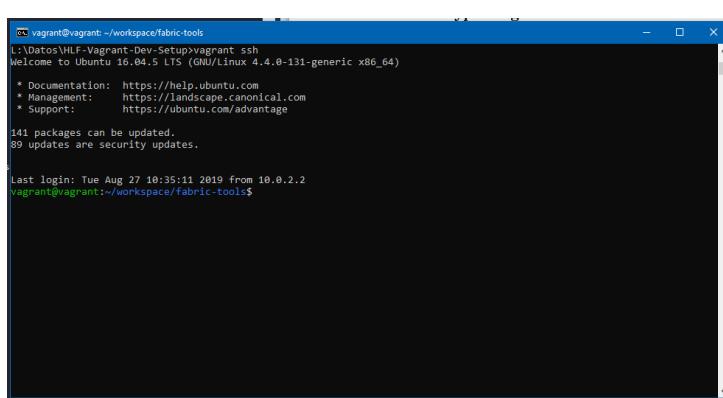
\$ vagrant up

Cuando la máquina haya arrancado ya estará disponible para ser configurada, para ello debemos acceder por terminal remoto (ssh) a la máquina virtual, Vagrant nos facilita esta tarea gestionando de forma interna y automática la comunicación por lo que solo deberemos ejecutar el siguiente comando para establecer la conexión:

\$ vagrant ssh



```
C:\WINDOWS\system32\cmd.exe
L:\Datos\HLF-Vagrant-Dev-Setup>vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
--> default: Checking if box 'bento/ubuntu-16.04' version '201812.27.0' is up to date...
--> default: Clearing any previously set forwarded ports...
--> default: Clearing any previously set network interfaces...
--> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
--> default: Forwarding ports...
    default: 8080 (guest) => 8080 (host) (adapter 1)
    default: 3000 (guest) => 3000 (host) (adapter 1)
    default: 2375 (guest) => 2375 (host) (adapter 1)
    default: 7050 (guest) => 7050 (host) (adapter 1)
    default: 7051 (guest) => 7051 (host) (adapter 1)
    default: 7052 (guest) => 7052 (host) (adapter 1)
    default: 7053 (guest) => 7053 (host) (adapter 1)
    default: 7054 (guest) => 7054 (host) (adapter 1)
    default: 5984 (guest) => 5984 (host) (adapter 1)
    default: 22 (guest) => 2222 (host) (adapter 1)
--> default: Running 'pre-boot' VM customizations...
--> default: Booting VM...
--> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
--> default: Machine booted and ready!
--> default: Checking for guest additions in VM...
--> default: Mounting shared folders...
    default: /vagrant => L:/Datos/HLF-Vagrant-Dev-Setup
    default: /home/vagrant/scripts => L:/Datos/HLF-Vagrant-Dev-Setup/scripts
    default: /home/vagrant/workspace => L:/Datos/HLF-Vagrant-Dev-Setup
--> default: Machine already provisioned. Run `vagrant provision` or use the `--provision` flag to force provisioning. Provisioners marked to run always will still run.
L:\Datos\HLF-Vagrant-Dev-Setup>
```

```
vagrant@vagrant:~/workspace/fabric-tools
L:/Datos\HLF-Vagrant-Dev-Setup>vagrant ssh
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-131-generic x86_64)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/advantage

141 packages can be updated.
89 updates are security updates.

Last login: Tue Aug 27 10:35:11 2019 from 10.0.2.2
vagrant@vagrant:~/workspace/fabric-tools$
```

Configuración de la máquina Vagrant

Estando dentro de la máquina virtual ya podemos proceder a la instalación de prerequisitos. Como dijimos, esta versión dispone de utilidades que facilitan esta tarea en gran medida, lo primero que debemos hacer es habilitar los permisos de ejecución de estas utilidades (si no los tienen ya).

```
$ chmod 755 ./scripts/*.sh
```

Tras esto ya podemos ejecutar las utilidades para instalar los requisitos de hyperledger, la primera instalará los requisitos (Python, Node.js, Curl, Docker, etc...) de los programas (requiere salir de la máquina y volver a entrar), la segunda instalará las utilidades de desarrollo de Hyperledger Fabric y finalmente la última instalará Hyperledger Composer.

```
$ ./scripts/install-prereqs.sh
$ logout
$ vagrant ssh
$ ./scripts/install-fabric-tools.sh
$ ./scripts/install-composer.sh
```

Instalación de Hyperledger Fabric

Con la máquina configurada completamente ya se puede instalar la implementación de la blockchain de Hyperledger (Hyperledger Fabric) para ello simplemente ejecutamos el siguiente comando para descargarlo e instalarlo:

```
$ ./downloadFabric.sh
```

Configuración del Host

Para mayor comodidad en las pruebas podemos configurar la máquina que alberga la máquina virtual (el Host) para que pueda gestionar los servicios de la misma, por un lado, que tenga acceso al servicio de contenedores Docker y por otro lado que pueda comunicarse con la blockchain para instalar BNA o gestionar credenciales.

Para lo primero se requiere, en el caso de Windows, añadir una variable de entorno para que indiquen al Docker del host donde debe conectarse. Esta es “DOCKER_HOST” la cual se debe asignar el valor “tcp://localhost:2375”, si no funciona también se puede poner la IP de la máquina virtual directamente. Se deberán borrar, en caso de que exista, las variables con el nombre “DOCKER_TLS” o “DOCKER_TLS_VERIFY”. Un ejemplo de configuración se puede ver en la **figura 21**.

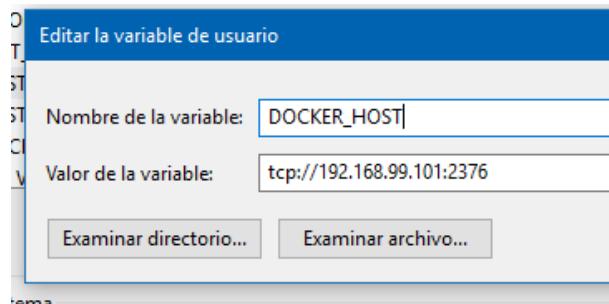


Figura 21: Variable de Entorno en Windows

Para lo segundo, deberemos instalar en la máquina Host los requisitos previos de Composer, estos son Node.js, Docker, Python 2.7. Tras eso podemos instalar los servicios de Composer, para ello ejecutamos los siguientes comandos:

```
$ npm install --global --production windows-build-tools
$ npm install -g composer-cli
$ npm install -g composer-rest-server
```

El primer comando es necesario en caso de que el segundo de error al no encontrar el compilador de Microsoft (recomiendo ejecutarlo igualmente), el segundo instalará el componente de línea de comandos de Composer y finalmente el último nos proporciona el cliente REST de Composer para nuestra blockchain.

El siguiente paso es el de creación una una “Card” en nuestro Host para poder autenticarnos con Fabric como administradores de la blockchain, para ello debemos ir al directorio “util” y ejecutar los siguientes comandos:

```
$ npm install
$ node devutil.js
```

Dentro del programa DevUtil debemos pulsar la opción “Install Peer Admin Card”, tras eso tendremos en nuestro directorio de usuario una carpeta que guardará las credenciales para la autenticación con el rol de administrador (PeerAdmin@hlfv1).

Es posible lograr esto directamente desde la máquina Host sin pasar por el “servidor” de la blockchain porque el directorio local de la máquina virtual que almacena todas las claves privadas de la blockchain es accesible por la máquina Host. El programa “devutil.js” busca esas claves para generar directamente credenciales válidas para la blockchain. Si hacemos lo mismo desde la máquina virtual obtendríamos el mismo resultado, pero tendríamos que exportar la “Card” a nuestro equipo local.

Iniciar Hyperledger Fabric

Con todo lo anterior configurado ya podemos administrar completamente nuestro entorno, desde la máquina virtual realizaremos las tareas de mantenimiento de las aplicaciones (imágenes Docker) y desde el Host realizaremos todas las gestiones y configuraciones de la blockchain. Para iniciar nuestra blockchain debemos lanzar las imágenes de Docker de la máquina virtual (los peers, la base de datos, el ordering peer, etc...) para ello simplemente ejecutamos el siguiente comando, el cual inicialmente descargará las imágenes del repositorio oficial de Docker:

```
$ ./startFabric.sh
```

Para parar el entorno de Fabric podemos usar el siguiente comando (borrará toda la configuración que se haya hecho en la blockchain):

```
$ ./stopFabric.sh
```

```
vagrant@vagrant:~/workspace/fabric-tools
XMMHPad0iyC497dnURa
-----END CERTIFICATE-----
2019-08-27 10:41:25.290 UTC [msp] setupSigningIdentity -> DEBU 035 Signing identity expires at 2027-06-24 12:49:26 +0000 UTC
2019-08-27 10:41:25.290 UTC [msp] Validate -> DEBU 036 MSP Org1MSP validating identity
2019-08-27 10:41:25.291 UTC [msp] GetDefaultSigningIdentity -> DEBU 037 Obtaining default signing identity
2019-08-27 10:41:25.291 UTC [grpc] Printf -> DEBU 038 parsed scheme: ""
2019-08-27 10:41:25.292 UTC [grpc] Printf -> DEBU 039 scheme not registered, fallback to default scheme
2019-08-27 10:41:25.292 UTC [grpc] Printf -> DEBU 039 ckecSolverWrapper: sending new addresses to ccc: [[peer0.org1.example.com:7051 0 <nil>]]
2019-08-27 10:41:25.292 UTC [grpc] Printf -> DEBU 039 ClientConn switching balancer to "pick_first"
2019-08-27 10:41:25.292 UTC [grpc] Printf -> DEBU 039 pickFirstBalancer: HandleSubConnStateChange: 0xc4202d4460, CONNECTING
2019-08-27 10:41:25.293 UTC [grpc] Printf -> DEBU 03d pickFirstBalancer: HandleSubConnStateChange: 0xc4202d4460, READY
2019-08-27 10:41:25.293 UTC [channelCmd] InitCmdFactory -> INFO 030 Endorser and orderer connections initialized
2019-08-27 10:41:25.294 UTC [msp/identity] Sign -> DEBU 03f Sign: plaintext: 0440870A5C0B0911A0C0B059194EB0051...E4B81F0178D01A0B80A090A000000000000A0
2019-08-27 10:41:25.294 UTC [msp/identity] Sign -> DEBU 040 Sign: digest: 53EAAADE88ED63C0B9A386C452E755119C02C890A4E0316000DC43A5969FE9CB
2019-08-27 10:41:25.292 UTC [channelCmd] executeJoin -> INFO 041 Successfully submitted proposal to join channel
vagrant@vagrant:~/workspace/fabric-tools$ ./stopFabric.sh
Developer mode only script for Hyperledger Fabric control
Running 'stopFabric.sh'
FABRIC_VERSION is set to 'hlfv12'
FABRIC_START_TIMEOUT is unset, assuming 15 (seconds)
Stopping peer0.org1.example.com ... done
Stopping couchdb ... done
Stopping org1.example.com ... done
Stopping ca.org1.example.com ... done
vagrant@vagrant:~/workspace/fabric-tools$
```

Si quisiésemos simplemente pausar el entorno de Fabric sin borrar nada debemos usar el siguiente comando, el cual forma parte de las utilidades extra que proporciona el autor del curso de Udemy.[10]

```
$ ./fabricUtil.sh stop
```

Tras eso podremos apagar la máquina sin miedo a perder nada, cuando queremos volver a iniciar la blockchain deberemos usar el siguiente comando:

```
$ ./fabricUtil.sh start
```

Anexo II: Creación de una Business Network Application

Para crear una aplicación en la blockchain, denominada en Hyperledger como Business Network Application (BNA), debemos hacer uso del Framework Yeoman el cual nos facilita mucho al crear un esqueleto de aplicación con un simple comando. Por ello lo primero es instalar Yeoman con el siguiente comando sobre la máquina Host:

```
$ npm install -g yo
```

Después debemos ejecutar Yeoman indicándole que queremos crear una aplicación de Composer como muestra el siguiente comando:

```
$ yo hyperledger-composer
```

Esto nos mostrará un asistente en el terminal que guiará la creación, al principio se tiene que indicar que lo que queremos crear una “Business Network”, el resto de opciones son meta-datos (nombre, licencia, autor, etc...). En la **figura 22** se muestra un ejemplo.

```
L:\Datos\yeoman>yo hyperledger-composer
Welcome to the Hyperledger Composer project generator
? Please select the type of project: Business Network
You can run this generator using: 'yo hyperledger-composer:businessnetwork'
Welcome to the business network generator
? Business network name: bna-test
? Description: Descripción de la bna
? Author name: Alberto Robles
? Author email: alberto.robles@um.es
? License: (Apache-2.0)
```

Figura 22: Asistente Yeoman para crear una BNA de Hyperledger Composer

Al terminar el asistente se creará un directorio con el nombre que le hayamos puesto a la BNA, dentro de él habrán tres directorios (lib, models y test) como se muestra en la **figura 23**, el primero tendrá un fichero JavaScript que contendrá la lógica de nuestra aplicación (los “contratos inteligentes”), el segundo tendrá los ficheros “.cto” que almacenan los modelos de datos de las entidades de nuestra blockchain (los assets, los tipos de transacciones, los roles y usuarios, etc...) y finalmente el último directorio almacenará las pruebas unitarias de nuestra aplicación.



Figura 23: Estructura del directorio de una BNA creada con Yeoman

Cuando tengamos completamente programada nuestra aplicación podemos compilarla e instalarla en la blockchain para ello debemos crear dentro del directorio de nuestra BNA una carpeta llamada “dist” y dentro de ella ejecutar el siguiente comando para compilar, instalar e iniciar la BNA en la blockchain:

```
$ composer archive create -t dir -n ../
$ composer network install -a <archivoBNA> -c PeerAdmin@hlfv1
```

```
$ composer network start -c PeerAdmin@hlfv1 -n <nombreBNA> -V 0.0.1
-A admin -S adminpw
```

```
L:\Datos\HLF-Vagrant-Dev-Setup\yeoman\iot-bna\dist>composer archive create -t dir -n ../
Creating Business Network Archive

Looking for package.json of Business Network Definition
Input directory: ../

Found:
  Description: Business Network app for IoT data storage
  Name: iot-bna
  Identifier: iot-bna@0.0.1

Written Business Network Definition Archive file to
  Output file: iot-bna@0.0.1.bna

Command succeeded

L:\Datos\HLF-Vagrant-Dev-Setup\yeoman\iot-bna\dist>composer network install -a iot-bna@0.0.1.bna -c PeerAdmin@hlfv1
✓ Installing business network. This may take a minute...
Successfully installed business network iot-bna, version 0.0.1

Command succeeded
```

Figura 24: Resultado de compilar e instalar una BNA

Tras eso tendremos en ejecución nuestra BNA sobre Hyperlegder Fabric, además, al ejecutar el comando de instalación se crea una “Card” para poder autenticarse como administrador de esa BNA (pero no administrador de Fabric), para importar la “Card” en nuestro banco de credenciales usaremos el comando:

```
$ composer card import -f <fichero.card>
```

Si fuese necesario borrar una ya existente se tendría que ejecutar el siguiente comando:

```
$ composer card delete -c <fichero.card>
```

```
L:\Datos\HLF-Vagrant-Dev-Setup\yeoman\iot-bna\dist>composer network start -c PeerAdmin@hlfv1 -n iot-bna -V 0.0.1 -A adminpw
Starting business network iot-bna at version 0.0.1

Processing these Network Admins:
  userName: admin

✓ Starting business network definition. This may take a minute...
Successfully created business network card:
  Filename: admin@iot-bna.card

Command succeeded

L:\Datos\HLF-Vagrant-Dev-Setup\yeoman\iot-bna\dist>composer card delete -c admin@iot-bna
Deleted Business Network Card: admin@iot-bna

Command succeeded

L:\Datos\HLF-Vagrant-Dev-Setup\yeoman\iot-bna\dist>composer card import -f admin@iot-bna.card
Successfully imported business network card
  Card file: admin@iot-bna.card
  Card name: admin@iot-bna

Command succeeded
```

Figura 25: Resultado de iniciar una BNA e importar la Card de administrador

Anexo III: Iniciar Composer REST con autenticación JWT

En este anexo se va a tratar el proceso para iniciar el servicio REST que proporciona Hyperledger Composer, además, se explicará cómo se activa la autenticación JWT.

Tras desplegar la blockchain e instalar la aplicación distribuida con éxito es necesario iniciar la API REST de Composer para que el servicio proxy se pueda conectar con la blockchain, para ello es necesario preparar los requisitos.

El primer requisito es el de una base de datos MongoDB, esta se puede instalar fácilmente con el instalador oficial. En la instalación no hace falta ninguna configuración específica, dejando todo por defecto es suficiente. La página para descargar el instalador es la siguiente:

<https://www.mongodb.com/download-center/community>

El siguiente paso es el de instalar la estrategia JWT en el módulo Passport.js, para ello se debe ejecutar el siguiente comando en la maquina Host:

```
$ npm install -g passport-jwt
```

Es necesario realizar unas pequeñas modificaciones en la estrategia, para ello se debe crear un fichero JavaScript (por ejemplo, “*custom-jwt.js*”) en el directorio de los módulos globales de npm (“%appdata%\npm\node_modules”) con el siguiente código:

```
1 const passportJwt = require('passport-jwt');
2 const util = require('util');
3
4 function CustomJwtStrategy(options, verify) {
5   options.jwtFromRequest = passportJwt.ExtractJwt.fromUrlQueryParameter("token");
6   passportJwt.Strategy.call(this, options, verify);
7 }
8
9 util.inherits(CustomJwtStrategy, passportJwt.Strategy);
10
11 module.exports = {
12   Strategy: CustomJwtStrategy
13 };
```

El motivo de modificar la estrategia original es para simplificar su funcionamiento por comodidad, en concreto, se simplifica la manera de obtener el token de sesión del cliente. En la línea 5 se muestra la nueva forma que se implementa, en ella el token se recibirá como un parámetro GET llamado “*token*” concatenado en la ruta.

Por último, antes de iniciar el servicio REST, queda la generación de la clave secreta y de un token de autenticación para el cliente (el proxy). La clave secreta puede ser cualquier cadena de 32 caracteres (256 bits), como ejemplo se usará “*mBikxDAzkmh7eC3yZUOq5ZIFHz9rhF9h*”.

Con la clave secreta definida ya se puede crear el JWT del cliente, para facilitar el proceso se hará uso de la página “*jwt.io*” en la cual se disponen de dos columnas, en la primera se tiene el JWT firmado (el que tendrá el cliente) y en la segunda se muestra la información decodificada.

En la **figura 26** se muestra la apariencia inicial de la página.

En la segunda columna se dispone de un formulario denominada *payload* el cual representa la información en JSON que se guardará y firmará en el JWT, en el ejemplo de la plataforma se

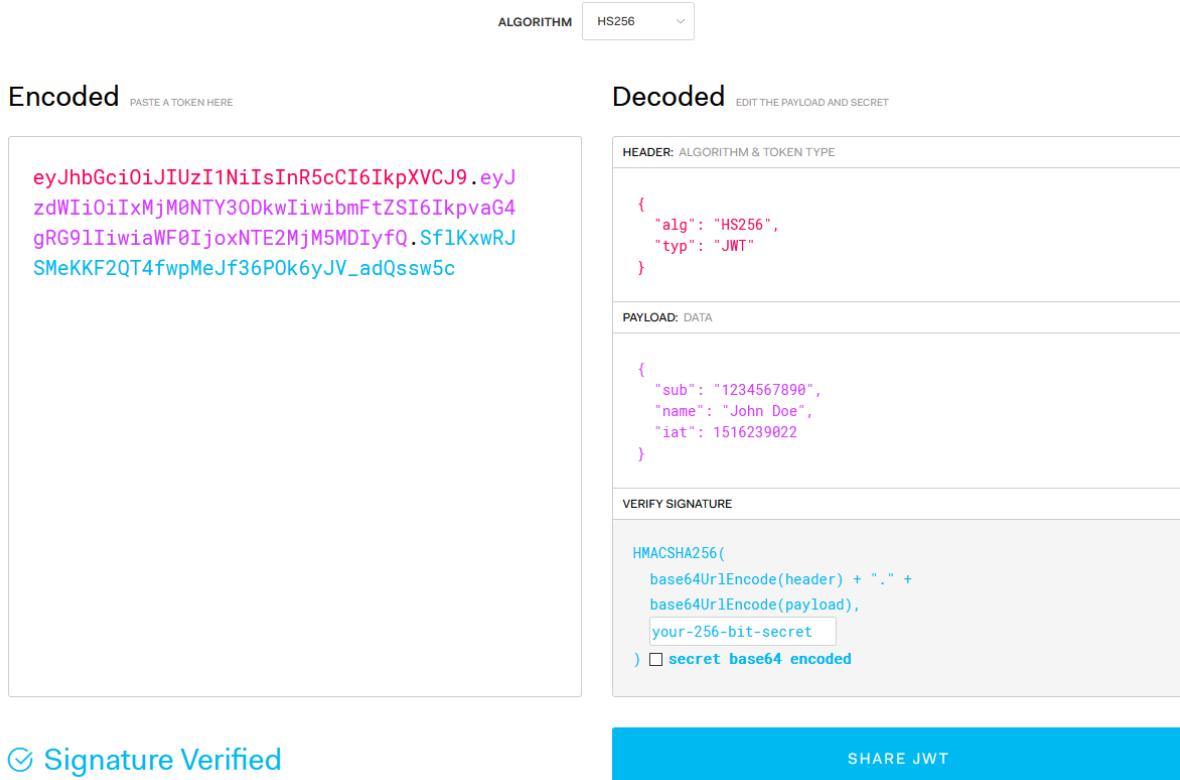


Figura 26: Página jwt.io

usan solo dos campos (marca de tiempo y usuario). En un escenario en producción se podría ampliar la información añadiendo firmas del usuario y más metadatos útiles para el proceso de autenticación.

El siguiente JSON muestra el ejemplo de *payload* de la plataforma:

```

1 {
2   "timestamp": 1555587358,
3   "username": "alberto"
4 }
```

El último paso es el de firma los datos, para ello se debe poner la clave secreta en el último formulario de la segunda columna, inmediatamente a eso la primera columna se actualizará mostrando el JWT final que en el caso del proxy es:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJ0aWlk3RhXAiOjE1NTU10DczNTgsInVzZXJuYW1lIjoiYWxizXJ0byJ9.
3SKZabM6pu5Ig6-zx_vA40y_QG6r2mi3dgoT8uLoY10

En la **figura 27** se muestra la apariencia de la página al configurarla con los parámetros del proxy.

Con todos los requisitos listos ya es posible iniciar el servicio REST, para ello se debe iniciar un terminal e ir introduciendo uno a uno los comandos que se explican a continuación.

El primero de ellos es el que indicará a Composer donde se encuentra la base de datos:

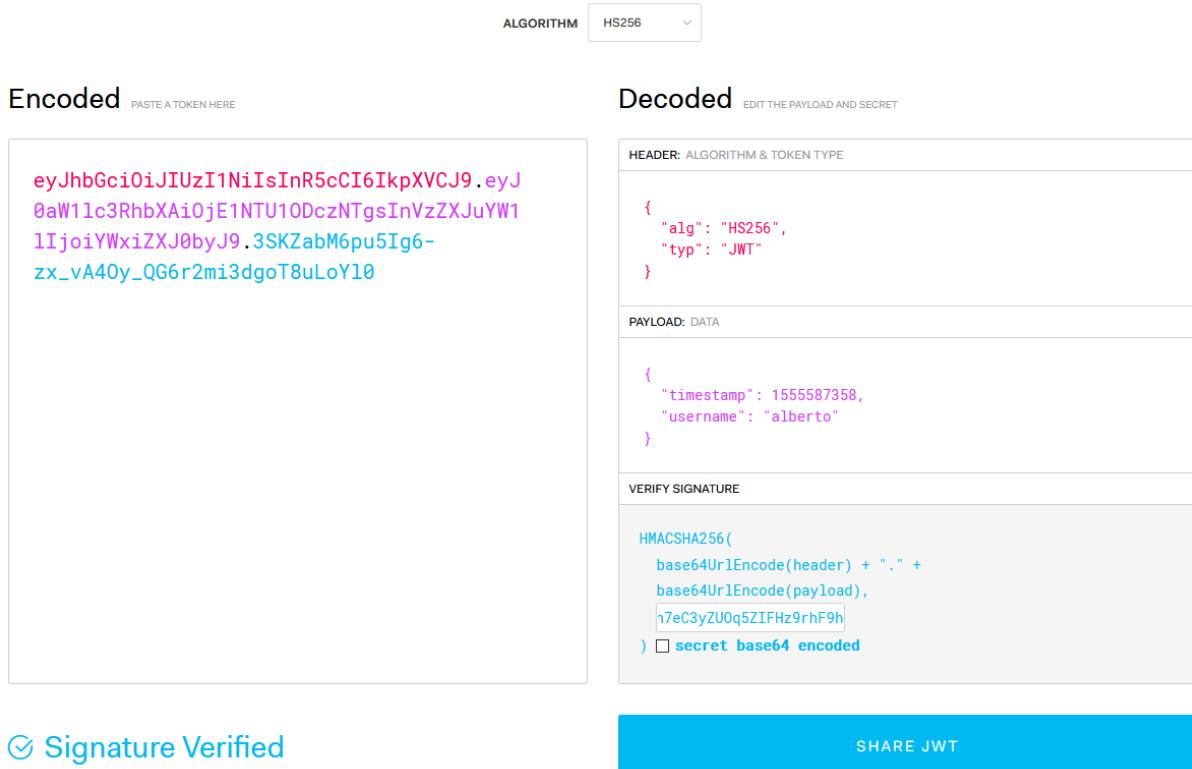


Figura 27: Página jwt.io con el ejemplo del proxy

```
$ SET COMPOSER_DATASOURCES={"db": {"name": "db", "host": "localhost", "connector": "mongodb" }}
```

Seguido a este vienen los referentes a la BNA, se indica la *Card* administradora y si usa namespaces:

```
$ SET COMPOSER_CARD=admin@iot-bna
```

```
$ SET COMPOSER_NAMESPACES=never
```

Se indica a Composer que requiere autenticación para aceptar operaciones:

```
$ SET COMPOSER_AUTHENTICATION=true
```

Se configura el mecanismo de autenticación, para ello se indica que será JWT con el fichero modificado y la clave secreta:

```
$ SET COMPOSER_PROVIDERS={"jwt": {"provider": "jwt", "module": "custom-jwt.js", "secretOrKey": "mBikxDazkmh7eC3yZU0q5ZIFHz9rhF9h", "authScheme": "saml", "successRedirect": "/", "failureRedirect": "/"}}
```

Finalmente se inicia el servicio en sí:

```
$ composer-rest-server
```

Con lo dicho ya se tiene disponible el servicio REST para ser accedido desde el proxy o desde cualquier navegador local al servicio.

Anexo IV: Iniciar nodo IPFS

La ejecución del nodo IPFS local es bastante sencilla en comparación con los otros anexos, únicamente es necesario instalarlo y ejecutarlo.

Para instalar el entorno IPFS se disponen de varias alternativas, en nuestro caso ya que es necesario un único nodo sin ningún tipo de interfaz se hará uso de Go-IPFS, este se puede descargar en la siguiente dirección:

<https://github.com/ipfs/go-ipfs/releases>

Para instalarlo, dado que no se da ningún instalador ejecutable, simplemente se debe descomprimir el fichero en la ruta que se quiera.

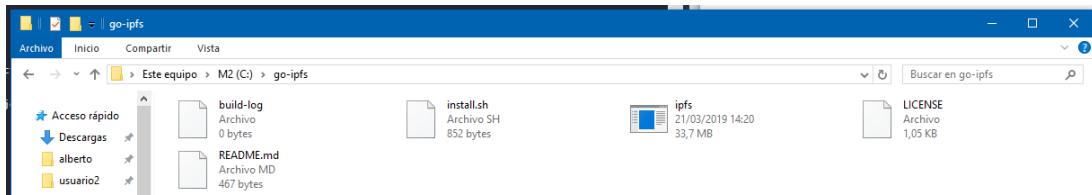


Figura 28: Directorio de Go-IPFS

Cuando se necesite tener operativo el nodo IPFS se tendrá que ejecutar en modo segundo plano, esto se logra ejecutando el siguiente comando en la carpeta del programa:

```
$ ipfs daemon
```

Como resultado de ese comando se mostrará una dirección web para usarla como servicio REST y además, esta sirve de panel de administración del nodo. En la **figura 29** se muestra el resultado del comando y en la **figura 30** la apariencia visual de la página de administración.

```
C:\WINDOWS\system32\cmd.exe - ipfs daemon
C:\go-ipfs>ipfs daemon
Initializing daemon...
go-ipfs version: 0.4.19-
Repo version: 7
System version: amd64/windows
Golang version: go1.11.5
Swarm listening on /ip4/127.0.0.1/tcp/4001
Swarm announcing /ip4/127.0.0.1/tcp/4001
API server listening on /ip4/127.0.0.1/tcp/5001
WebUI: http://127.0.0.1:5001/webui
Gateway (readonly) server listening on /ip4/127.0.0.1/tcp/8181
Daemon is ready
```

Figura 29: Resultado de la ejecución del nodo IPFS

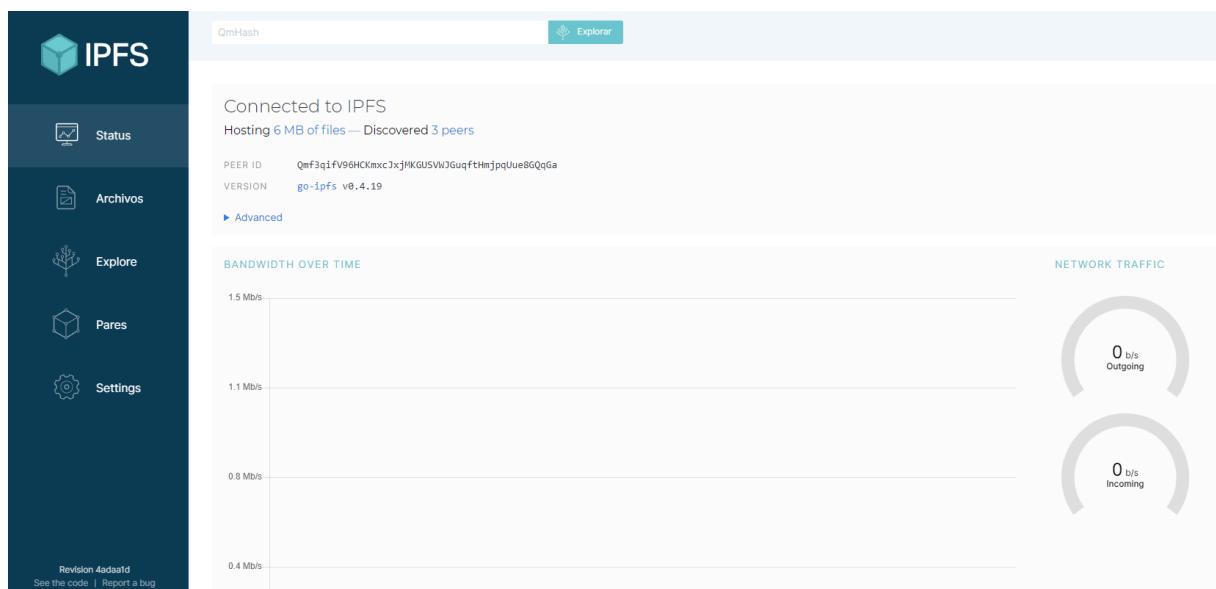


Figura 30: Panel de administración del nodo IPFS

Anexo V: Iniciar Servicio Java y probarlo

La solución propuesta se ha implementado en Java, dentro de un proyecto Java Web. Para iniciar la plataforma se debe configurar adecuadamente el entorno Eclipse.

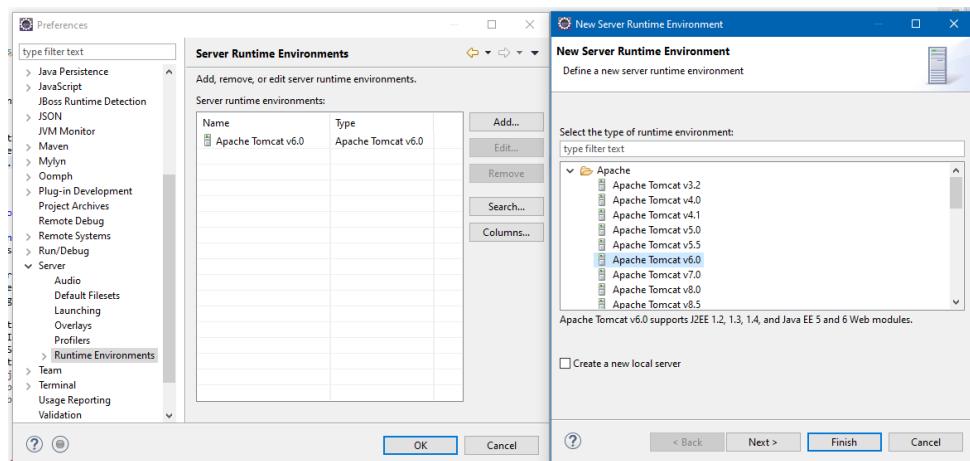
Lo primero que se debe configurar es el servidor de despliegue, para la implementación realizada se ha usado Apache Tomcat 6, este se puede descargar en la siguiente dirección:

<https://archive.apache.org/dist/tomcat/tomcat-6/v6.0.32/bin/>

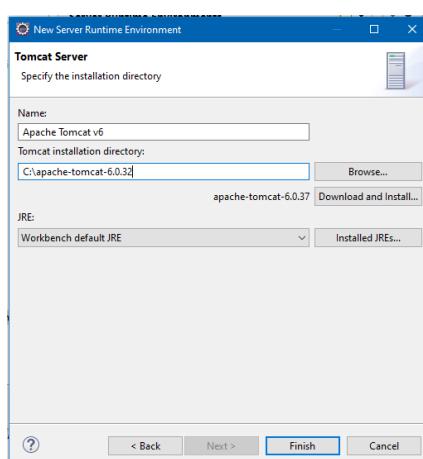
Lo que se descarga es un fichero comprimido que se debe descomprimir en el directorio que se quiera (se recomienda en el raíz del disco principal).

Seguidamente se debe añadir a Eclipse el servidor de despliegue y configurar el proyecto para que lo use. Primeramente se debe crear el servidor apache, para ello se debe ir a la pestaña “Ventana” y dentro darle a “Preferencias”, después ahí buscar la opción “Entornos de ejecución”.

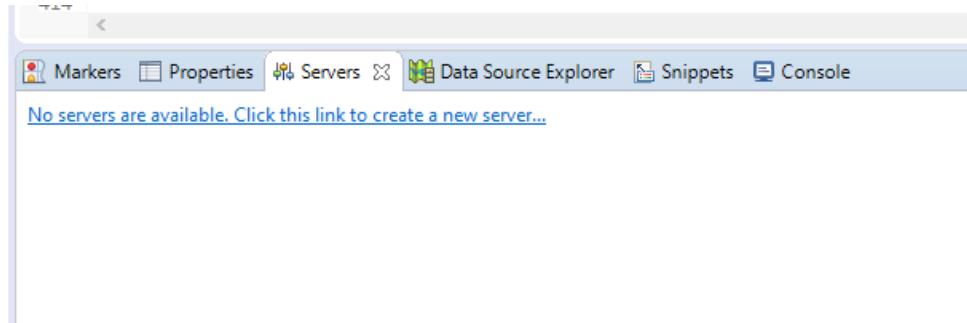
Tras entrar saldrá una lista de los entornos de ejecución creados en el proyecto, se debe dar a “Añadir” para crear unos nuevos, esto abrirá una ventana en la que se debe buscar la versión de Apache 6.0.



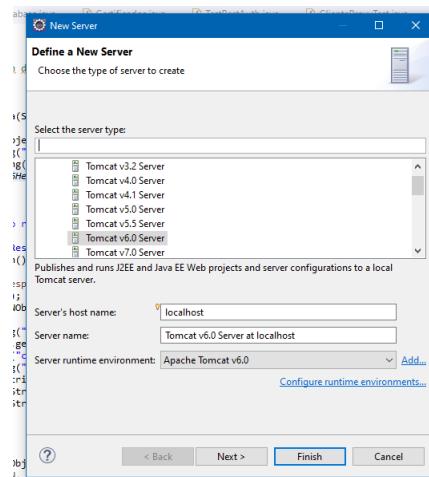
Después, se debe dar al botón “Siguiente” lo que cambiará la ventana a una donde se pide la ruta del servidor y el nombre que le queremos dar.



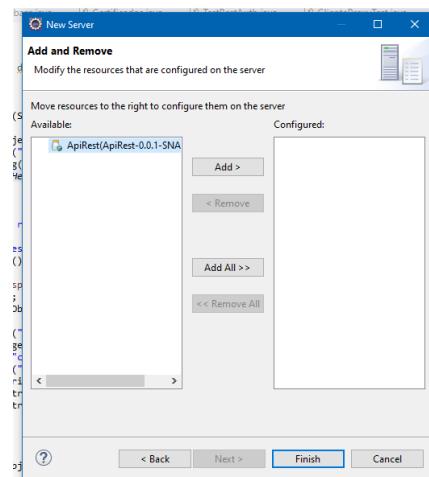
Ahora se debe configurar el servidor recién creado en el proyecto actual, para ello en la parte inferior del entorno se debe buscar una pestaña llamada “Servidores”, en ella inicialmente no saldrá ninguno y saldrá un mensaje que servirá de asistente para configurarlos.



Al pulsar el texto se abrirá el asistente de configuración, este pedirá la versión de servidor a configurar y el servidor instalado que se desea usar.



Finalmente pedirá indicar que proyectos se desean desplegar en el servidor, para ello simplemente se selecciona el proyecto y si le da al botón de añadir.



Con lo dicho ya se tendría el entorno de ejecución configurado adecuadamente para las pruebas,

para ejecutarlo se debe arrancar el servidor de despliegue y acceder a la ruta por defecto del proyecto ([“`http://localhost:8080/ApiRest/iot-bna/`”](http://localhost:8080/ApiRest/iot-bna/)).

Para realizar las pruebas de funcionamiento se puede hacer uso de cualquiera de los clientes programados, en concreto el más simple es el que se diseñó para hacer las pruebas, simplemente se debería ir a la clase “ClienteProxyTest” y ejecutarla, tras eso mostrará en la consola el resultado.

Además, si se quiere una prueba más realista se puede ejecutar la clase de ejemplo IoT “ClienteIoTApp”, de esta forma se puede probar si la creación de transacciones de forma periódica funciona adecuadamente.

The screenshot shows the Eclipse IDE interface with the following details:

- Java Project:** apirest - Java EE - Apirest/src/test/ClienteTapp.java - Eclipse
- File Menu:** File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, Help
- Quick Access:** index.html
- Project Explorer:** Shows the project structure with files like ConectorREST.java, IoTRestApp.java, Database.java, Certificados.java, TestRestAuth.java, ClienteProxyTest.java, ClienteTapp.java, and index.html.
- Code Editor:** The main editor contains Java code for a REST API test. It includes imports for java.util, java.net, javax.ws.rs, org.json, and org.junit.*. The code uses Jersey's Client API to make requests to the local host. It handles responses, logs them, and performs assertions. A try-catch block handles exceptions, specifically for certificate errors.
- Console:** Shows the command line output of the test execution. It includes the Java command, the class name, and the timestamp (2019-07-20T09:58:53). The output shows the test running, connecting to the server, sending requests, and receiving a successful JSON response from the /logon endpoint.

Anexo VI: Instalar la aplicación cliente en una Raspberry

Para probar el cliente Java en un dispositivo IoT Raspberry se debe exportar la aplicación y ejecutarla en el dispositivo, para ello lo primero que es necesario es instalar el sistema operativo en la Raspberry.

Existe una distribución de Linux, mantenida de forma oficial, para las Raspberries llamada **Raspbian**, en la página oficial se puede descargar directamente:

<https://www.raspberrypi.org/downloads/raspbian/>

Además, ofrecen varias guías para instalarlo desde cualquier sistema operativo. Las guías se pueden consultar en el siguiente enlace:

<https://www.raspberrypi.org/documentation/installation/installing-images/README.md>

De forma resumida el proceso de instalación del sistema operativo consiste en descargar la imagen oficial, obtener una tarjeta SD y volcar la imagen en la tarjeta haciendo uso del programa adecuado (en función del sistema operativo) para configurarla como una partición de arranque. Tras eso se podrá encender directamente la Raspberry con la SD, mostrándose la pantalla de acceso inicial.

Con el dispositivo ya funcionando, y con la configuración de red adecuada, se puede proceder a instalar la máquina virtual de Java, para ello se hará uso del gestor de paquetes que trae el sistema operativo pero añadiéndole otra fuente de paquetes:

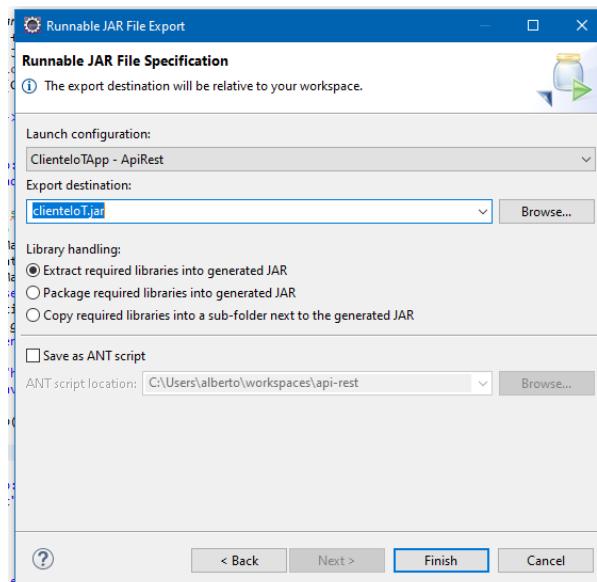
```
sudo apt-key adv --recv-key --keyserver keyserver.ubuntu.com EEA14886
```

```
sudo echo 'deb http://ppa.launchpad.net/webupd8team/java/ubuntu trusty main' >> /etc/apt/sources.list
```

```
sudo echo 'deb-src http://ppa.launchpad.net/webupd8team/java/ubuntu trusty main' >> /etc/apt/sources.list
```

```
sudo apt-get update
```

```
sudo apt-get install oracle-java8-installer
```



Finalmente quedaría exportar la aplicación Java del proyecto Web, para ello en Eclipse debemos darle a “Archivo”, y después a “Exportar”, dentro del asistente se debe buscar “Archivo JAR Ejecutable”. A continuación, se debe seleccionar el módulo de ejecución principal (“ClienteIoTApp”) y el destino.

Finalmente, se debe copiar el fichero JAR al dispositivo IoT, por ejemplo con SCP. Para ejecutarlo habría que escribir el siguiente comando en el terminal del dispositivo:

```
java -jar clienteIoT.jar
```




UNIVERSIDAD DE MURCIA
FACULTAD DE INFORMÁTICA

Middleware de seguridad para Blockchain en escenarios IoT

Autor
Alberto Robles Enciso

