UNIVERSITÀ DEGLI STUDI DI TRENTO

Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

A blockchain-based solution for logistics tracking

Supervisor

prof. Montresor Alberto

Student

Melotti Damiano

Academic year 2018/2019

Acknowledgments

The first thanks goes to Prof. Montresor, for his guidance and constant support during the work of this thesis. His knowledge and patience have been fundamental to achieve a good result. I would also like to thank the Spindox Labs team, for giving me the opportunity of studying and exploring these technologies. I am thankful to my family and my girlfriend, for being always by my side and supporting me in every moment of these years. Finally, a special thanks to Filippo, who has been not only a roommate, but also a valuable friend and an endless source of inspiration and advice. His presence accompanied me during my bachelor studies and he totally deserves my gratitude.

Contents

Summary

All logistic processes involve the exchange of goods between entities that do not fully trust each other. For example, let us consider a very common scenario: a delivery by a courier that includes an intermediate stop. This can be the porter of a building, or an authorized post office. What usually happens is that when the parcel arrives, the receiver testifies with his/her signature (on paper or on an electronic device) that the exchange has occurred. However, this is not always true, because not all the deliveries include this possibility or because the persons directly performing the exchange do not fully respect the protocol. As a result, the process is vulnerable: a hand-written signature is often counterfeitable and when it is not even present, it is possible to generate a debate "my word against yours".

This problem becomes worse when it includes more steps: this happens, for example, in complex systems of organizations, people and activities employed in the supply chain sector: here the handovers are even more articulated and it is important to guarantee a successful outcome of the processes, that are crucial for the companies involved. A series of signatures can be a solution, but, as aforementioned, several things can go wrong.

Who has not been taken into consideration yet is the final recipient. They know nothing about the events happening and they could only be notified autonomously by one of the entities involved, whom they would have to trust.

During the spring 2019, I completed a three-month internship at Spindox (Trento); together with their R&D team, named Spindox Labs, we investigated a solution to this problem. We started from the wide experience that the company had with trackers, trying to understand if one of them could be applicable to this case. IoT devices are spreading more and more thanks to the many applications they have. Both industry and research are very active in this area, hence we had to conduct an in-depth analysis to explore the various alternatives available. We eventually found a good solution, that meets our requirements for communication, connectivity and geolocation. After this, the most significant part of our research involved the study on the architecture that our system required. We wanted to find an innovative solution that eliminated any centralized authority, because did not want to introduce one in the system: many entities at the same level, without a supervising one.

The result of this work is a decentralized architecture approaching the problem with a combination of diverse technologies. We implemented a private blockchain, to guarantee distribution and reliability, we integrated IoT devices to track the deliveries and we included web servers and interfaces to facilitate the usage of the service. This project has been a research activity, thus no specific requirements from any client have been considered. The solution is a "proof-of-concept", demonstrating the power of the technologies involved and it can represent a complete starting base for a customized version, tackling a particular problem.

The dissertation is organized as follows: Chapter 1 and 2 introduce the problem and present the architecture. Then the proposed solution is split into its three main components, namely tracking, decentralization and servers. Chapter 6 evaluates the project, outlining limits and possible extensions. Chapter 7 concludes the thesis.

1 The scenario

This chapter introduces the problem, giving a more detailed description. In fact, we have only given a few examples in the Summary, but we need a precise model of the use case. Then we focus on decentralization, explaining what the main characteristics of such approach are and defining some concepts.

1.1 Problem statement and solution architecture

Formalizing the problem, the typical scenario is the following. We have a sender S and a final recipient R. S wants to send R a particular good, G. To perform the exchange, S will designate a certain route that involves n intermediate stops $(I_1, I_2, ..., I_n)$ and will instruct m dealers $(D_1, D_2, ..., D_m)$ that will take care of each part of the delivery. From a theoretical point of view, nothing forbids to have two identical dealers D_i - D_j. Indeed, as we will see later, we are not really interested in the dealers themselves, but more in the companies that these dealers work for. We assume that S decides the route or, at least, knows the various stops needed.

We would like to find a model able to register precisely each handover, by making use of secure techniques that do not allow counterfeiting or fraud: each entity should be protected against a potential malicious one. Finally, R should be able to monitor the events about G, in order to check the progresses.

The proposed solution will make use of IoT devices to track the location of the dealers. Additionally, we will define a protocol to use the connectivity of these trackers to detect other devices owned by the intermediate points. In this way, the verification will be automatized and, from the human point of view, we will only need a little expedient from the dealers. Furthermore, the architecture will not present any centralized element: the solution will be fully decentralized.

1.2 Decentralization

Unlike traditional architectures, that usually present a single element providing all the information for the service and practically having control, a typical distributed network shares knowledge between many nodes, therefore excluding a central point.

Centralization versus decentralization is a battle that does not have a definite winner. The Internet has seen several attempts to decentralize some services, like DNS and social networks, but the truth is that, in many situations, the benefits of keeping a central entity often outperform those of the correspondent distributed alternatives [17]. However, there are still some cases in which a P2P network is a valid solution, especially when there are hard requirements like privacy, anti-censorship and scalability. Despite their advantages, in fact, centralized architectures always present a single point of failure, with all its consequences.

When in 2008 the mysterious Satoshi Nakamoto published the Bitcoin paper [19], a particular type of distributed network became extremely popular: the *blockchain*. The author proposed "a purely peer-to-peer version of electronic cash", with the aim of introducing a payment system based on a crypto-currency, without intermediaries but still reliable and secure against typical attacks such as double spending. This problem may arise for two reasons. The first is that crypto-currencies have no watermarks to prevent the unauthorized production of money: consequently, it is in principle possible to copy one coin and spend it twice. The second is a direct disadvantage of a distributed system: propagation delays have to be handled in order, to successfully distinct and validate the correct transactions. More generally, distributed networks need to implement a protocol to establish consensus between nodes. Bitcoin combines existing technologies in a sophisticated way, to solve its fundamental problems. Without examining in depth all these mechanisms, let us concentrate on how transactions are stored: using a blockchain.

The blockchain itself is just a distributed data structure: as the name suggests, it is composed by

blocks and each block contains transactions. These transactions are defined with a precise grammar, that obviously depends on the type of application. In Bitcoin, for example, transactions respect the (simplified) form "the result of transaction T_i is used as input to send x BTC to A, with fee y", where A is the public key of the recipient. Beyond the grammar, to implement a blockchain there is the need of a function to validate transactions. Evidently, the previous sample transaction will be refused, if the sender does not have the rights to use the output of T_i. More precisely, Bitcoin uses a measure called proof-of-work, in order to not only include a control on the actual validity of the transactions, but also to require a strong computational effort to approve a block. This is a crucial element to improve reliability and contrast the possibility of subverting the network with Sybil attacks.

What makes the blockchain different from other data structures are two properties. First, it is distributed: every node of the network has its own copy of a *ledger*, that is kept consistent and up to date with gossip protocols. Second, the way in which these blocks are linked: each one contains the hash fingerprint of the previous. As a result of these, once data is inserted in a block it impossible to change it. In practice, changing a transaction would mean recalculating the hash value of all the blocks succeeding the modified one, replacing the fingerprint and convincing all the network that the new copy is the one that should be used. This is impossible, unless one controls more than half of the overall computing power. Immutability is the first and fundamental property of the blockchain, that makes it a good solution to store money exchanges: obviously, crypto-currency designers do not want to allow the possibility of modifying any of the validated transactions.

The Bitcoin system showed the world the great potential of the blockchain. After its success, a number of other similar crypto-currencies were launched, with the so-called *initial coin offerings*, generating a new business for investors and speculators. A lot of hype is still surrounding this technology, that in any case presents some disadvantages and it is surely not the right solution for every problem [20].

Analyzing again our case, however, it seems to suit well. Handovers can correspond to the transactions, the goods can be our "coins" and the entities of the system can be the participants of the network, submitting exchanges. Essential requirements such as reliability and durability of transactions would be guaranteed by design. In Chapter 4 we discuss better this approach.

2 The solution

Figure 2.1 illustrates the solution that we have designed. Let us explain its characteristics, by first analyzing the overall flow and then concentrating on each step.

The notation used in the schema is the same as the one introduced in Section 1.1: from left to right, we notice the sender S, the good G, two dealers D_1 and D_2, two intermediate points I_1 and I_2 (each one with a responsible person at work) and finally the receiver R. Obviously, this scenario is just a possible combination of entities and the solution is not fixed to this one.

The following is an abstract description of the various actions that are part of the model. For this moment we are ignoring some specific details (e.g. the protocols used for the interactions, the format of the packets etc.), that are properly explained together with the singular components.

S starts the process by inserting into the system the information related to G. They have a dedicated server to provide the requested data. In particular, they need to declare the intermediate points and the dealers. This data will be stored in the blockchain as a new *asset*, something similar to the coins in crypto-currencies.

Then, the delivery is ready to start. D_i (the procedure holds for both D_1 and D_2) carries an IoT device tracking its location and searching for other BLE devices nearby. On the other side, the intermediate stop has a beacon, constantly advertising its UUID (Universally Unique IDentifier). The protocol for the exchange is the following:

- The dealer arrives at the destination, the tracker detects the stop and sends to a server a packet containing the beacons' identifiers found;

- The server controls if one of the identifiers corresponds to the one owned by the intermediary and, if so, sends a confirmation to the blockchain;
- The person in control of I_j collects G and sends a confirmation, using a web interface, to a separate web server.

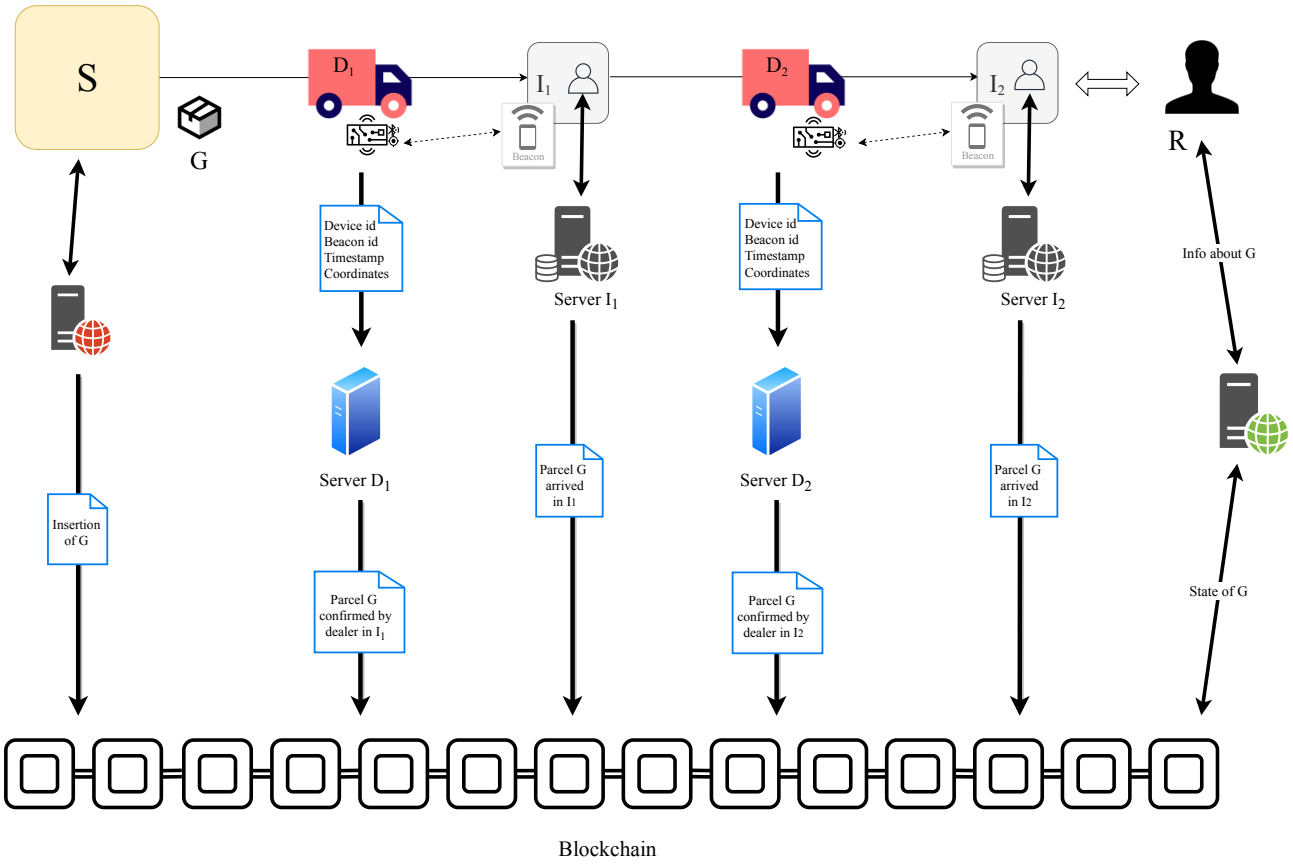

Figure 2.1: The schema of the discussed solution

The model as it is so far cannot be deceived by the dealer, as they have no control over the device; however, it is vulnerable to an attack from a malicious actor controlling I. They can, in fact, collect G and not inform the system about the exchange. As a result, we have lost track of who has the good. To solve this, there is the need for a protocol on the human side. Not complicated, but essential: the dealer has to make sure that who receives G really sends their confirmation to the server.

The final recipient, at any time, can access an interface that, by consulting the status of G on the blockchain, informs him about the progress of the delivery. They can check which exchanges happened and whether they succeeded. In case of problems it is immediate to find the responsible. Assuming that the protocol has been fully respected, if G is only confirmed by the dealer, it means that it is still in their hands; to the contrary, if only the intermediary confirmed the good, an error must have occurred, because there is no proof that the dealer really arrived at that place.

After this brief analysis there are a lot of unknown elements: how the devices work, how the communications take place, how data is actually saved. In the next chapters we discuss these aspects, dividing the project into the three main branches that constitute it.

3 Tracking

The first part that we analyze is the tracking. Our solution needs devices that support geolocation, bluetooth and can connect to the Internet to send messages. Clearly these are just the basic requirements, additional features can be useful in future developments. Another important aspect is the battery, since these devices cannot ensure long autonomy if frequently active.

First, we give a brief description of the modern technologies for the communications of IoT devices, then we dive into the market to analyze the various options.

3.1 Communication technologies

The state-of-the-art of these networks is very wide and presents several alternatives, each of them with advantages and disadvantages. They can be divided by many parameters, since many variables need to be taken into account when modelling the architecture of an IoT project. Our use case requires a standard able to guarantee the delivery of packets at arbitrary distance, consuming as little battery as possible. These requirements drive us to a new wireless communication technology: LPWAN (Low Power Wide Area Network).

LPWAN is increasingly becoming popular thanks to its low-power, long-range and low-cost communication properties. It is particularly suitable for IoT applications that only need to transmit small amounts of data in long range and for its specifications it is preferable to traditional cellular options (such as 4G and LTE networks).

Many LPWAN technologies have been developed in both the licensed and unlicensed frequency bands. LoRa, Sigfox and NB-IoT are today the leading ones, with many technical differences [18].

LoRa

LoRa (Long Range) is a patented digital wireless communication technology that was first developed by Cycleo - a French start-up - and then acquired by Semtech (USA). Its ecosystem can be divided in two parts, LoRa and LoRaWAN: the latter is the standard protocol for WAN communications and the former is used as a wide area network technology. In other words, LoRa is the physical layer and LoRaWAN is the MAC and application layer of the stack.

LoRaWAN provides various classes of end devices to satisfy different requirements. It ensures a better battery life compared to the other LPWAN technologies, but this also implies lower data rates and longer latency. To connect to the Internet, LoRa requires a gateway. The infrastructure is the following: end devices communicate with one or more LoRa gateways, which then forward messages to a cloud, a network server or to another gateway. On the contrary, when a message is sent to a device, the network chooses the best gateway.

One important aspect of LoRaWAN is determined by the so-called *duty-cycle*: defined as the maximum percentage of time during which an end-device can occupy a channel, is a key constraint for networks operating in unlicensed bands. Thus, each device has a threshold limiting the overall transmission time. However, the amount of time that a device will need is unknown, since it depends on the distance between the end point and the nearest gateway. The farther the gateway, the longer it will take to deliver a message. As a result, the total amount of messages sent is unknown and totally unpredictable considering our use case, in which we cannot make assumptions on the distribution of the gateways (w.r.t. the location of the tracker) [12].

Sigfox

Sigfox is a patented technology that was developed in 2010 by the start-up SigFox. It is an LPWAN network operator that uses a wide-reaching signal called “ultra narrow-band”. This system presents

a significant link asymmetry: a downlink communication, i.e., data from the base stations to the end devices can only occur following an uplink communication and in every case the number of messages is limited to 4 per day. On the uplink, instead, the amount of packets sent cannot be more than 140 per day, with a maximum payload length of 12 bytes. These limitations are similar to the LoRa's ones: in fact, they were included by the designers for the same reasons. Clearly, acknowledgements cannot be supported and consequently the reliability is ensured using time and frequency diversity as well as transmission duplication.

NB-IoT

NB-IoT is a radio technology standard developed by 3GPP (3rd Generation Partnership Project) to enable a wide range of cellular devices and services. It is based on the LTE protocol, therefore guaranteeing the high performance level associated with cellular connections, but at the cost of more complexity and greater power consumption [23]. While other infrastructures have gateways that aggregate sensor data, which then communicate with the primary server, with NB-IoT sensor data is sent directly to it. For this reason it is being touted as the potentially less expensive option.

LTE-M is another LPWAN radio technology standard based on LTE. However, we did not take it into consideration because there is a lack of coverage and support from operators in Italy.

3.2 The market

The market of the IoT is constantly growing. There are numerous applications for these devices, both indoors and outdoors, with a variety of different requirements. For this reason, almost every company offers more than one solution, to best meet the needs of the buyer. We proceed now with the analysis of some candidate trackers. Table 3.1 summarizes their technical features.

Company	**Device**	**Connectivity**	**GNSS**	**Customizable firmware**	**Additional**
Estimote	LTE beacon	Bluetooth 5.0, LTE-M/NB-IoT, NFC	GPS, GALILEO, GLONASS	No, but scripts can be included	accelerometer, temperature, LED, programmable button
Accent Systems	IoT tracker	Bluetooth 5.0, Wi-Fi, LTE-M/NB-IoT	GPS, GLONASS, Galileo, QZSS, BeiDou	No	accelerometer, temperature, LED, buzzer
Sensolus	Ultra	BLE, Wi-Fi, Sigfox, GPS	GPS	No, but cloud API available	accelerometer, temperature
Pycom	FiPy + PyTrack	Bluetooth, Wi-Fi, LTE-M/NB-IoT, LoRa, Sigfox	GPS, GLONASS, Galileo, QZSS	Yes	accelerometer, LED

Table 3.1: Specifications of the devices

Estimote, LTE beacon

Estimote, Inc. is an American company that offers mostly beacons of different types [3]. Their main focus is in indoor applications, but they also offer a device particularly close to our needs: the LTE beacon.

This tracker can connect to the Estimote cloud over LTE. Through this platform it is possible to include JavaScript code that will be run by the device, but the firmware itself is not customizable. This is the first negative point: every communication of the device is directed to its cloud and it is not possible to bypass it. Furthermore, the LTE beacon comes with a subscription including 1000 syncs per year per each device (for example, about 2-3 syncs per day). There's also a data-cap of 100 MB per year.

The tracker offers a good range of sensors and built-in radios. On the other hand, though, Estimote claims that, if frequently synced with the server, the battery life will be shorter than 2 days. Consequently, choosing the LTE beacon would imply the need of an external power bank or a permanent connection to the vehicle.

Accent Systems, IoT tracker

Accent Systems designs and develops tailored IoT devices, offering an end-to-end solution [1]. Like Estimote, they have a wide range of beacons and one particular device for outdoor tracking via LPWANs, called IoT tracker.

This solution is not personalizable in any way. The firmware is not modifiable and it communicates with a platform called Inmolecular (over LTE), a built-in ecosystem to show the tracker's data. The device, as specified in the website, is more destined to be integrated in an asset or in a vehicle to track only its location, not really to send custom packets as our project requires. Nevertheless, its characteristics make it a valid alternative.

Sensolus, Ultra

Sensolus is an industrial IoT company, based in Belgium [11]. Their main offer is addressed to logistic companies that want to track their assets. Indeed they propose an easy and ready-to-go service, similarly to the Accent Systems' one. Ultra is the tracker that we considered as a potential solution.

This is another device not open to personalization. However, the cloud exposes numerous APIs to read data sent by the trackers: the overhead is still present, but our server could coexist. Concerning the battery, the presence of a replaceable one is a double-edged sword, because on the one hand it is can ensure longer life, but on the other it requires an action to change it.

This device only supports Sigfox connectivity: the APIs are actually an extension of those provided by the operator. As pointed out in the previous section, this communication technology has some limitations in the number of messages that can be sent, which inevitably affect this tracker too.

Pycom, Fipy and PyTrack

Pycom offers hardware products that vary from plug and play devices to single components such as development boards and accessories [8]. There are two separate products that together can form a good tracking solution: a development board (FiPy) and an additional module to support geolocation (PyTrack).

This alternative is completely different from the other ones: instead of offering an entire environment, almost ready to use, this device comes completely empty and needs to be programmed. Even the battery is not present, so one can decide which one to use, or directly power it via cable.

3.3 The choice

Analyzing the various alternatives outlined, we opted for the Pycom device. The first reason is the possibility of fully programming the firmware, which guarantees a high level of flexibility. Indeed, we need a complete control on the device: the algorithm needs to be elaborated and written by us. With all the other solutions, this would not possible; building our service on top of another one was not ideal.

In addition, the FiPy board offers a wide range of communication technologies that we could use: from the simple Wi-Fi for the developing phase to all the mentioned LPWANs. As for the latter, we identified NB-IoT as the best technology. Similarly to the previous decision, we made this choice

because, despite the higher battery consumption, it offers a solid, reliable and unrestricted network, which can be used without creating a particular infrastructure or having to limit the number of messages.

3.4 Firmware

As mentioned before, the chosen device allows us to program its firmware completely. The programming language to be used is MicroPython, an implementation of Python 3 optimized for integrate controllers or constrained environments [6]. It includes a subset of Python libraries, together with some modules specific to the MicroPython implementation.

Figure 3.1 describes the device's life cycle. After booting, it attaches to the LTE network and initializes the Bluetooth and GPS sensors. Then, it tries to find the coordinates until it succeeds (actually, in the real implementation, a check is added, to handle cases in which it is impossible to locate the tracker; in other words, there is a maximum number of attempts). After the BLE scan, it checks whether the connection is still on: this is necessary because the dealer might have arrived in an area where there is no signal. Successively, it sends the data to the server, or it stores it in an internal file. Clearly, when the file is not empty and there is connection, the device will send the saved packets too. Finally, it turns off for a specific number of seconds.

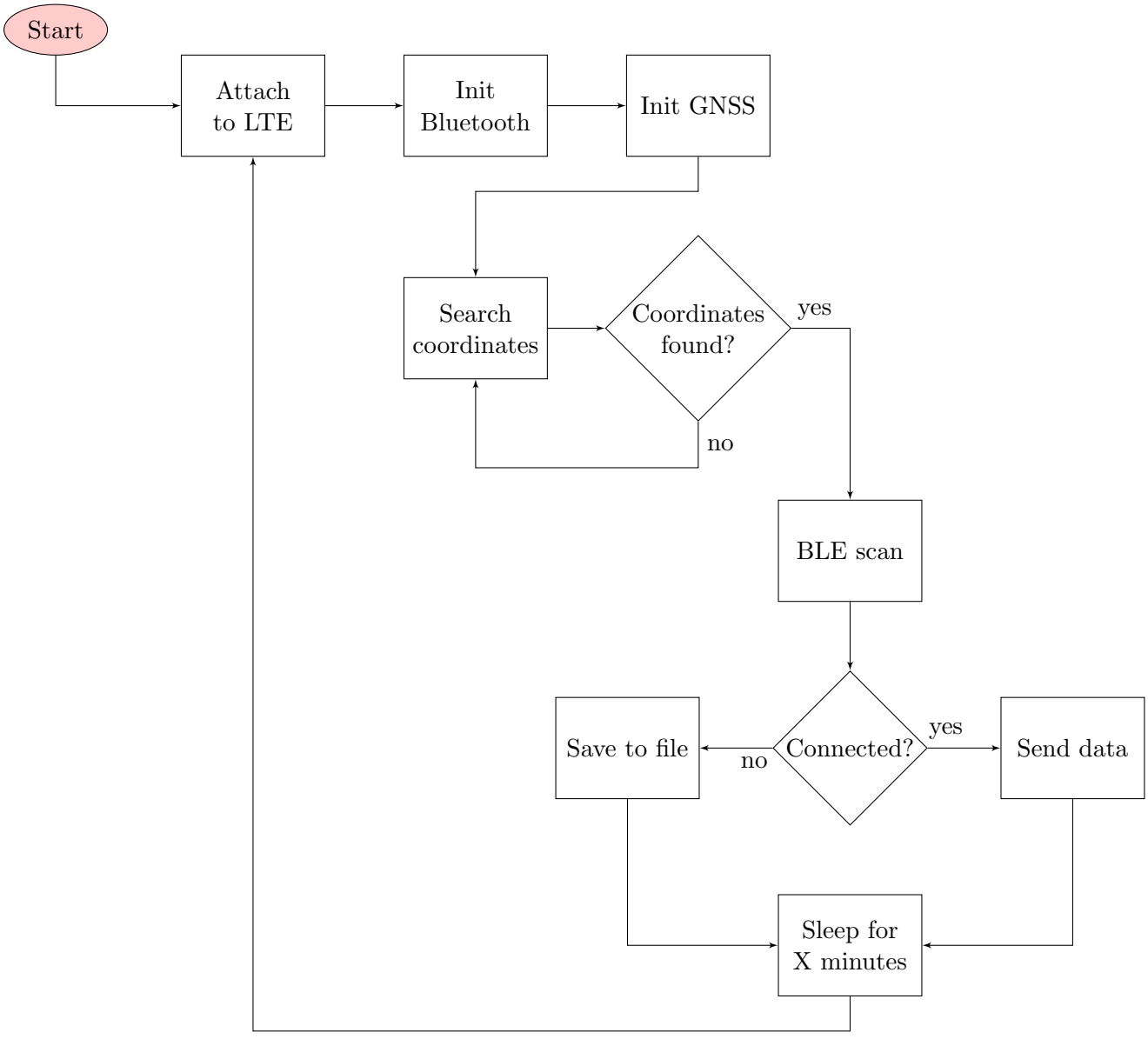

Figure 3.1: Flowchart of the tracker's algorithm

The program is basically an endless cycle, the device is always on or in stand-by. It is important to switch off all the sensors before the latter phase, because the battery needs to be preserved as much as possible. In every "working" phase, the device is restarted completely, thus there is no difference with respect to an initial boot: this is a very useful feature because every iteration is independent from the others.

Battery consumption is often the first concern in the development of software for IoT devices. Every single instruction has to be written in the most efficient way that the language offers, especially if it interacts with sensors. The battery is also the only reason for which a device could stop working: if it runs out, the tracker will inevitably turn off. In addition to a smart usage to preserve it, we used the device's LED light to warn the dealer when the percentage is below a certain threshold. In this way, they will know when it has to be recharged. Anyway, the practical integration of the device needs to be defined: in particular scenarios, it could be always in charge, plugged to a vehicle port. In other ones, there may be a necessity for long battery life, because no power supply is available. In both cases, as described in Section 3.2, the device can be adapted by using a different battery. This is another good reason that justifies our choice.

Concerning the development environment, Pycom has developed an IDE plugin, to make programming their modules as simple as possible. Named *Pymakr*, it is available for two popular text editors (Atom and Visual Studio Code). Pymakr establishes a communication towards the device and enables a REPL (Read-Eval-Print Loop) console to execute MicroPython commands directly on it. Using the plugin it is also possible to upload an entire firmware, modify the existing one or reset the device.

The firmware has to include two fundamental files: `boot.py` and `main.py`. When booting the device they are executed automatically. Usually, the first is used to prepare the connectivity and perform the needed preliminary actions, while the second is the algorithm itself. Obviously, additional files can be included, to be used as libraries or as support resources. In particular, we added a configuration file, in order to adapt the instructions to the use case. Through this resource, which is implemented as a simple JSON dictionary, several parameters can be set: for example, the number of seconds that the device waits in sleep mode, the number of attempts to retrieve and send the coordinates, the battery threshold for warning, information about the server receiving data (such as IP address, port, protocol). Considering the real scenario in which the device will be integrated, it will be possible to obtain a different battery life depending on these parameters.

The updates are sent using MQTT. The communication part is reported in detail in Section 5.2.1, that analyzes the back end. However, let us briefly describe how packets are composed. The single updates are JSON strings, that contain a very limited amount of information: the coordinates of the device, the list of bluetooth identifiers detected and a timestamp. We decided not to include additional data to keep the packets as small as possible. Nonetheless, all the necessary is present: the coordinates give the location of the tracker, the UUIDs will be parsed to register a potential delivery and the timestamp is useful if the message is not sent immediately. When there is no signal, in fact, these strings are saved in a file and will be sent later.

Being in a limited environment, not all the libraries are available by default: if a particular one is necessary, it has to be included as an ad hoc file, in the `lib` folder. Pycom offers a set of built-in libraries, especially developed for their devices. Beyond them, many more modules are available: the standard Python libraries have been "micro-ified" to fit in with the philosophy of MicroPython. They provide the core functionality of that module and are intended to be a drop-in replacement for the standard Python library. All the mentioned resources are freely distributed and can be download from the public repositories of MicroPython and Pycom [7] [9]. Examples of libraries that we have actually used are:

- `ujson`, to read the configuration file and save the unsent messages - this an example of a Python standard module that has been simplified and lightened by MicroPython;
- `L76GNSS`, for geolocation;
- `umqtt`, to use the tracker as MQTT publisher;
- `deepsleep`, to disable the device's features and set the stand-by mode.

4 Decentralized architecture

After the individuation of the IoT devices, the second fundamental part of the project is the architecture. We already introduced in Section 1.2 the intention of using a distributed network, possibly implementing a blockchain. However, there are different solutions that make use of this data structure to create a distributed ledger, that need to be analyzed. Unlike the previous sections, however, this time the decision is more straightforward.

4.1 Decentralization alternatives

There are two main alternatives for a blockchain-based architecture: permissionless and permissioned [15]. In a permissionless blockchain, such as the cryptocurrency ones, anyone can be a node of network, write in the shared ledger by proposing new transactions (that have to be validated) and participate in the process to reach consensus. In this scenario, the absence of trust within the participants is typically mitigated by introducing the concept of *mining*, a computational expensive process required to validate transactions. The term derives from the compensation given to the nodes that manage to approve a block, in form of transactions fees or new coins.

In contrast, permissioned blockchains are composed by known entities, which can be members of a consortium or stakeholders in a given business context; different methods are implemented to control and update the state, and there can be particular strategies to limit who can issue transactions. In this case, the risk of a malicious behaviour from a participant is considerably lower, because the guilty can be identified more easily. The possibility of controlling the participants of the network, together with the better performances in terms of latency and throughput, obtained by avoiding computational intensive consensus, render this type of blockchain more attractive for large corporations.

In our use case, the choice is simple: we have a precise set of participants that have a common goal, we want to make data available only to the nodes that need it and we do not plan to include a mining process. A permissioned blockchain is absolutely the best solution. Also concerning the framework, we did not consider many alternatives. There is one that perfectly suits our needs: Hyperledger Fabric.

4.2 Hyperledger

Hyperledger is "an open source collaborative effort created to advance cross-industry blockchain technologies" [4]. In other words, it is a group of related projects and tools, started by the Linux Foundation and supported by many IT companies (such as IBM and Intel), with the goal of offering blockchain-based solutions to build innovative applications. Under the Hyperledger "umbrella", there are several frameworks that range from permissionless to permissioned networks, with different targets. In particular, Fabric is the most interesting project for us: it offers a distributed ledger platform designed for industries, with a versatile and highly configurable architecture. It supports even pluggable consensus protocols and identity management protocols, enabling a full extensibility basing on the use case.

In addition, the framework is free, easy to use, install and deploy. More specifically, the Fabric documentation provides the images to run every peer of the network in Docker containers. Docker is a tool designated to create, deploy and run applications in containers [2]. Unlike a Virtual Machine, that creates a whole virtual operating system and inevitably implies a significant overhead, a container is just a set of processes that are isolated from the rest of the system. They run from a distinct image that provides all files necessary to support them. A single machine can run several Docker containers much more easily than VMs, because all of them share the host kernel. These characteristics of isolation and efficiency are very important for our project, because the servers need to run more than one application (as described in Chapter 5) and performances are fundamental. Using Docker, it is enough to use different containers for the various services deployed. Finally, Docker is also open-source, thus it is possible for everyone to add more features not available by default.

To start building a network, Hyperledger provides a set of tools. In fact, it would be rather difficult to configure a network by working directly with low level container images and deployment details. As follows, we mainly used the tools Composer and Explorer.

We have introduced a lot of elements that need to be explained in details. In the following sections, we describe accurately how Hyperledger Fabric works, how it is composed and how we built our network.

4.3 Hyperledger Fabric structure

In this section we report and explain the main building blocks of Hyperledger Fabric [13].

Assets

Assets are the virtual coins exchanged in the Fabric environment. They can represent anything for which transactions are made. More specifically, they can be real products, like cars or houses, or intangible goods, like intellectual properties.

Concretely, assets are represented in the system as collections of key-value pairs, stored in JSON dictionaries. Every update of the assets is saved in the ledger.

In our system, as outlined previously, the assets are the goods being delivered. Their state is altered during the exchanges, through the submission of transactions, but the final recipient is always able to check every modification occurred.

Smart contracts

Smart contracts are another concept related to a blockchain network: in general, they are self-executing scripts that are triggered by some conditions. For example, Ethereum - another popular permissionless blockchain [14] - allows developers to program their "autonomous agents", using a particular DSL (Domain Specific Language), Solidity. This language is Turing-complete, hence smart contracts are concretely real programs, that can perform a wide range of actions. They can be used, for instance, so that funds are spent only when all the members of an organization (or a fixed percentage of them) give their authorization.

In the Hyperledger environment, a smart contract defines the business logic that controls the lifecycle of the assets. It is also called *chaincode*, interchangeably, even though the former is the transaction logic itself, while the latter is the container of a group of contracts which is then deployed in the network.

In our use case, smart contracts are used to manipulate the state of the goods. The functions are executed on the ledger's current state and are triggered by a transaction proposal. The result is a set of key-value pairs that can be applied to the ledger and distributed over all peers.

Ledger

Focusing now on the ledger itself, it can be described as the list of transactions maintained by each member of the network. It is composed by two parts. First, the *chain*, which is the log of all transactions that have been made, kept in the tamper-resistant blockchain with all the properties outlined in Section 1.2. Second, the *world state*, representing the latest values for all the keys.

The current state is very important because it is the layer directly involved when transactions are submitted. To improve efficiency, it is stored in a state database (such as LevelDB or CouchDB), which is basically an indexed view of the transaction log. It can always be regenerated from the chain.

The ledger contains also configuration blocks defining essential information such as access control lists and policies. We explain better these concepts in Section 4.4, to report how we used them.

Peers

Being one of the main reasons for which we chose this framework, the first characteristic of Fabric is that it is private and permissioned. The members of a network, indeed, have to enroll via an MSP

(Membership Service Provider). This guarantees a complete control of the infrastructure and the possibility of identifying directly all the participants.

More precisely, each organization is responsible for setting up their peers. Each of them hosts an instance of the ledger and an instance of chaincode. As a result, redundancy is specifically introduced, to avoid single points of failure. Unlike public blockchains, though, the various peers have different roles, which we list here. In the following section, a typical transaction flow helps understanding better how these roles come into play.

Orderer peers: the central nodes that group transactions in blocks and delivers them to all the other peers. They represent the direct source of consensus.

Endorsing peers: the specialized peers designed to validate transactions. Fabric introduces them to increase scalability.

Anchor peers: the discoverable peers that receive updates from the the orderers and broadcast them inside their organization.

General peers: ordinary peers, that can submit transactions from their client applications. They basically act as proxies to connect clients to validating peers.

It can be pointed out that an orderer peer is potentially a single point of failure, thus exposing a Fabric network to the same problems of a traditional centralized architecture. If there is only one orderer node, this is true, but a real solution (in production) should consider this issue and avoid it. This point is better clarified in the consensus section.

In our project, it is reasonable to have a single peer per organization acting as orderer and endorser. Unlike other cases, in which the network requires a higher number of endorsing peers, because many transactions are not valid, in our model the interactions are more controlled (by design of the servers, see Chapter 5) and a balanced approach is a good one. This has the advantage of reducing overhead, because both operations are unified in the same peer, but at the same time it requires special protection for those nodes, since they are essential for the organizations. General and anchor peers can be deployed based on the requirements of the entities: if the number of interactions is particularly high, they can consider adding more general peers, to improve load balancing and availability of the service; on the contrary, if an organization does not have these specific requirements, it can concentrate everything in one node, representing the anchor and general peer.

The different type of peers, concretely, are deployed as Docker containers. As a result, ideally, a whole Fabric network could be implemented on the same machine. As mentioned before, the co-existence of various containers is made possible by their design. Clearly, a real network has to be distributed, but nothing forbids to have a particular host being both a general and anchor peer, or an orderer and endorser. In every case, it is essential to avoid single points of failure: a blockchain network is well implemented only if it is not vulnerable against the potential problems of centralized architectures. Otherwise, it is just an unjustified complication.

Channels

A channel can be simply defined as a virtual tunnel by which the members can connect with each other. Inside a single Fabric network, organizations can have multiple channels, regulated by specific policies. Peers need to authenticate to participate (via the Membership Service Provider) and there have to be an ordering service implemented to guarantee the functioning of the subnet. Each channel has its own ledger, thus communications are made private within the participants.

This is the main reason for which channels are important. In fact, they provide privacy not only between channels, but also towards the whole network. In this way, organizations can decide what they want to share with all the members and what they want to keep private between a subset of them. Even though this mechanism might seem contradictory, it is evident that different groups within the network will have a need for different information processes to be shared. Channels provide an efficient and reliable way to satisfy such requirement.

In our skeleton solution we did not include special channels, only the main one. However, they can be appropriate if, for example, the intermediate points are competing with each other, hence they do not want all deliveries to be visible publicly. In this case, an ad hoc channel could be created and the various organizations involved would be the participants.

Transactions

We can summarize the typical Fabric workflow in six steps. Here we consider a complete network with all the types of peers deployed separately. Figure 4.1 illustrates the protocol graphically; for the sake of readability, anchor nodes have not been included, so the ledger is directly sent to the general peers.

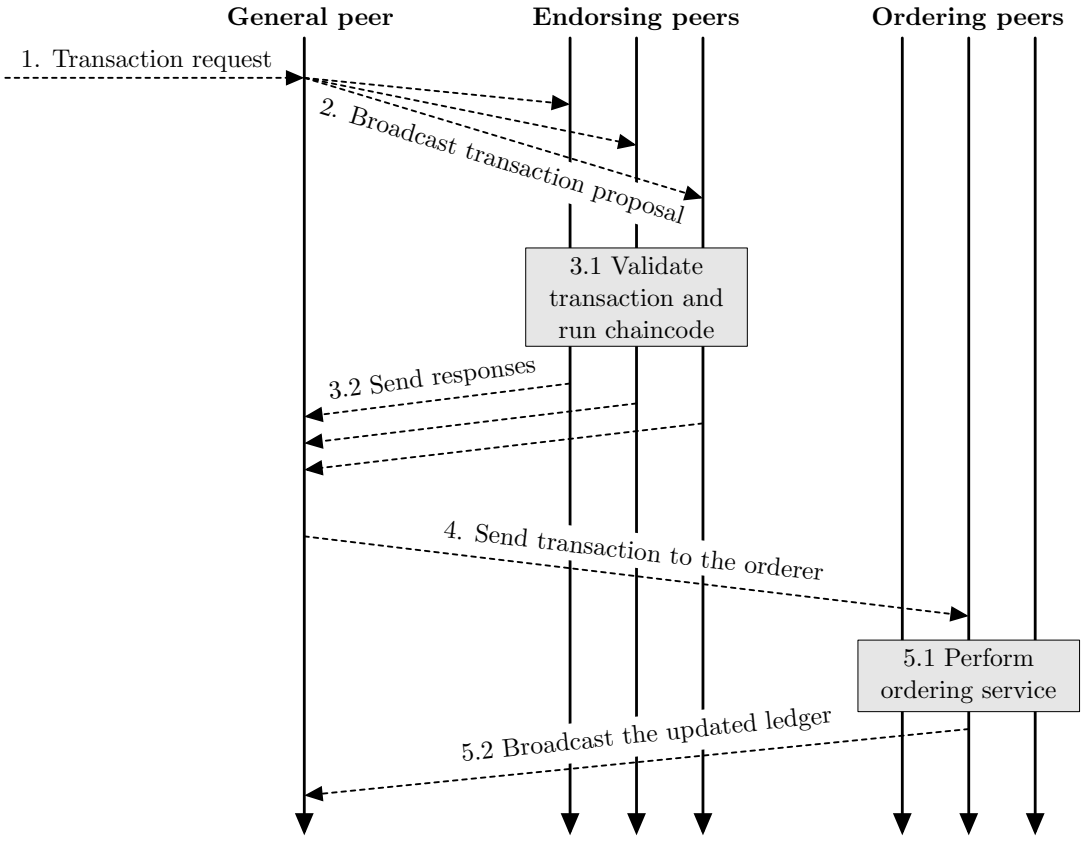

Figure 4.1: Schema of the transaction flow

1. A general node invokes a transaction request. This can be done through a client application.

2. The general peer broadcasts the proposal to the required set of endorsing peers. How they are chosen depends on the endorsement policy defined for a chaincode, but intuitively it will be composed by the endorsing peers of each organization involved in that transaction proposal.

3. Each of them checks the details needed to validate the transaction and runs the chaincode, independently. After that, they generate a response, containing the approval or the rejection of the proposal.

4. After receiving the needed number of positive responses, the peer sends the approved transaction to an orderer, to include it in the ledger.

5. Ordering nodes receive transactions from many different application clients concurrently. Their task is to work together to collectively form the ordering service, arranging batches of submitted transactions into a well-defined sequence and packaging them into blocks. After this, they have to forward the new block to the anchor nodes.

6. Anchor nodes finally forward the block to the other peers inside their own organization. These individual peers then update their local ledger.

Consensus

Consensus is a fundamental problem in distributed computing: in a nutshell, it consists in reaching an agreement among a collection of distributed processes. Reaching consensus in a distributed system, under the hypothesis of faulty nodes and unreliable communications, is impossible, as demonstrated by Fischer et al. [16]. However, a good distributed system always exhibits stable intervals of correct functioning during which consensus is achieved, in practice.

In Hyperledger Fabric, the consensus problem is approached with the introduction of orderer nodes. Unlike permissionless blockchains, in which every node can participate in the consensus process and the algorithms involved guarantee consistency at a high probability, the ordering service ensures deterministic consensus. That is, in Ethereum or Bitcoin it is possible the generation of divergent ledgers, namely *forks*, because the order of accepted transactions is not the same. A possible cause is the case of two (or more) miners forging different blocks at roughly the same time. These blockchains still have mechanisms to contrast this vulnerability, accepting only the longest chain and abandoning the so-called *orphaned blocks*. However, the risk exists, while in Fabric once a block is generated it is guaranteed to be correct and final by design.

Consensus algorithms also enable the nodes of a network to survive the failures of some of its members. These failures can be divided in two types: Byzantine faults and crash faults. In Byzantine faults, malicious nodes try to manipulate the other ones and affect the consensus process. On the other hand, crash faults are typically system crashes, network issues, packets loss etc. As a result, a consensus protocol can be CFT (Crash Fault Tolerant) or BFT (Byzantine Fault Tolerant). Intuitively, the latter can also tolerate crash faults, since Byzantine faults are practically a superset of all faults existing in distributed systems.

Fabric presents three different built-in implementations of protocols to achieve consensus in ordering service nodes.

- Solo, as the name suggests, features only one orderer node. Obviously, it is not fault tolerant, as it presents a centralized and vulnerable element. It cannot be a valid approach in production, but it can be useful when testing the architecture, because, from the other elements perspective, transactions are processed in the same way as with more complicated protocols.

- Raft is a CFT protocol that has been introduced in v1.4.1 [10]. The underlying model includes a dynamically elected node which acts as a leader, that receives all the updates, elaborates them and distributes its decisions among the others. In short, Raft nodes can be in three states: leader, candidate or follower. To elect a leader, the nodes set a random timer and candidate themselves when the timer is up. The candidate then broadcasts "vote request" messages and when the majority of the other nodes respond the leader is elected. To keep the configuration, Raft uses an heartbeat protocol and another timeout, that triggers a new leader election in case of faults.

- Kafka is another CFT protocol that has been adapted for use as an ordering service in Fabric [5]. Its logic is conceptually similar to the one used by Raft. However, the management, coordination and control of the cluster is handled by a Zookeeper ensamble, which is particularly complicated to set up: this is the reason for which Raft has been introduced by the Fabric designers.

It is worth noting that no BFT protocols are currently available, so the system is not capable of reaching agreement in the case of malicious nodes. However, as mentioned in the previous section, Fabric supports pluggable consensus: this means that a developer can include a different ordering service to manage the blockchain. Sousa et al. implemented a BFT ordering service for this framework [22]. This protocol is based on the BFT-SMART state machine replication/consensus library, with some optimizations to make it process up to ten thousand transactions per second, even with peers located at large distances.

Why Hyperledger does not offer by default a BFT ordering service is understandable: permissioned blockchains have a higher level of trust between their entities compared to permissionless ones. Participants are known to each other and all actions carry a signature of the organization submitting them, thus the designers preferred developing CFT-only protocols.

4.4 Development tools

For the development phase, Hyperledger offers some powerful tools, that help modelling and testing an architecture before actually deploying it. In our project we used two of them. Composer enables modelling of business networks and its resources; it makes easier the generation of a REST API and a UI application that allows for quick integration within an existing system. Explorer, instead, operates at a lower level and presents a dashboard with an overview of the calls available to query the blockchain or invoke transactions.

4.4.1 Hyperledger Composer

When building a new network it is advisable to start from this tool. The interface, called *Playground*, is very immediate and the documentation explains well how to use the various features offered. It is possible to access it directly from a web page, without installing anything, or loading the local version, that requires the development environment already set up.

This tool helps designing a network at a very high level of abstraction. From the Playground interface, there is no reference to endorsement policies, ordering services or other complex structures that we have outlined previously. Consequently, the creation of a network is very straightforward; it is easy to approach the technology and explore the various possible configurations, without considering implementation details.

To deploy a business network in Composer there are some files that need to be created, in order to define the network properly.

Model file

The model file declares the various components of the network: the resources - namely participants, transactions, assets and events - and the support structures, such as enumerated types and concepts. We did not mention events previously, because they are concepts directly related to Composer; they can be emitted by the tool and subscribed to by external applications. They are triggered when a transaction is committed.

The file is written using an object oriented modeling language. Concretely, the resources are classes with fields that can be of primitive types or references to other classes. It is required to set a field as unique identifier. For transactions and events, instead, identifiers and timestamps are automatically included. Our `model.cto` contains the definitions of four participants (the senders, the dealers, the intermediate points and the recipients), two transactions (confirmation from the dealer and confirmation from the intermediate point) and finally one asset (the goods exchanged).

Script file

The script file is necessary to define the transaction logic for the business network. The functions created are executed when a transaction is submitted: referencing Section 4.3, it represents the chaincode.

A processor function is the logical operation related to a transaction defined in the model. It usually modifies one or more values contained in the assets and updates the registry. If required, it can emit an event. Unlike the model file however, the scripts are written in Javascript, a complete and powerful language that offers all its constructs to handle, for example, asynchronous code or external API calls.

In our project, `script.js` simply implements the updates of the state of the asset when confirmations occur. Just a few checks are added to avoid the case of multiple confirmations from the same party and for the same good (which would be useless).

Access control list

This file defines the access control rules for business network. There are two types of access control: for resources and for network administrative changes. The evaluation follows a white list approach: if there is no rule granting a certain access, then it will be denied.

Each rule has to follow a precise grammar: it needs to have a description, a group or a particular participant (reported with the name inside the namespace), an operation (CREATE, READ, UPDATE, DELETE), a target resource (again using the namespace) and an action (ALLOW, DENY). Optionally, there can be conditions and transactions, defining the ones that the participant must have submitted in order to perform the specified operation.

In our network, the access control file (`permissions.acl`) is quite elaborated: we had to write a rule for participant to allow them to submit their transactions and to see the history of their operations only. Moreover, we denied the possibility of confirming a good from all the entities: even though this is a remote case, we do not want anyone but the participants directly involved to submit transactions about a certain delivery.

Optional - Queries file

This is a file with the purpose of defining some queries to retrieve information from the blockchain world-state. It is an optional component of the business network definition.

All queries must contain the description and statement properties. The former is a simple string, which can be used to explain in human language the function of the query, the latter is the query itself. The statement is written in an SQL-like query language and can accept runtime parameters. In our project we did not include this file.

At this point we have all the elements that qualify a network. The model gives us the structure, the scripts are executed when transactions are submitted and everything is controlled by the access control rules. The missing part is how to actually test a network. Composer provides a test section, to create participants, assets and transitions. Everything is done "behind the scenes", by clicking on a graphic interface. On the one hand, this level of abstraction is perfect to rapidly test the various files, but on the other there is the need to switch to a more concrete environment and test the real calls. That is why Hyperledger offers Explorer.

4.4.2 Hyperledger Explorer

After setting up a network using Composer, there is the possibility to deploy a "Composer REST server". Only in this phase it is possible to specify some particular aspects of the network, e.g. endorsement policies. This server can successively be used to communicate with the blockchain, simulating a real deployed network. Explorer operates between the server and developer, providing an interface with an overview of the functions available.

Developers can check information like name, state and list of network nodes, details of blocks, transactions and related data, chaincodes and any other relevant feature that may be stored on the blockchain. Since this data is usually in a format that is difficult to read for humans, this tool attempts to provide an easy visualization by using graphs, charts, pictures, and templates, in addition to the usual search and monitoring facility.

In parallel, it is also possible to connect to the REST server and test the various actions available. There are some standard operations offered by default: e.g., for every resource, it is possible to use a `GET` request to retrieve the JSON objects of all the entities, or of a specific one by including its identifier. Also `POST` requests are offered, to add new instances. In addition to these, all the user defined queries are reported in the interface as well.

This tool is very helpful to integrate these calls in already existing servers. It is enough to check the APIs, format the requests and use them from another program. In the next chapter we notice how immediate this has been.

5 Servers and Interfaces

The last major part of the project is the development of the various servers. The architecture presents four types of them, as listed below.

Sender server: connects to the blockchain to add new assets with their features.

Dealer servers: receive messages from the trackers and submit transactions with the confirmation from the dealers.

Intermediate point servers: offer an interface to confirm deliveries, submit transactions with the confirmation of the exchange points.

Receiver server: retrieves information regarding the progresses of a delivery and displays it to the user.

All the servers have been created using the same technology, Express.js. Express is, de facto, the standard server framework for Node.js. It offers a robust set of features to develop web applications. Being extremely simple to set up, it is the best choice to build the minimal web servers that we need.

The following sections describe singularly each server and the messages exchanged. It is necessary to specify that the details reported correspond to an embryonic version, a first implementation not intended for production. As mentioned in the Summary, the final solution is not deployed yet, thus we concentrate only on the necessary logic. In particular, the programs temporarily use the API provided by the Composer REST server to communicate with the blockchain: in the final version, the destination of these calls will change, but the functioning will stay the same.

5.1 Sender server

The first server that we analyze is the sender server. It represents the single service that needs to provide an interface to insert a new good in the system and to specify its route. When the sender connects to the web server, they need to know exactly the details of the delivery: in this first version, the dealers and intermediate stops are reported using their identifiers.

Concentrating on the back end part, this server receives the POST request of the user and generates another POST request, directed to the blockchain. Transporting a JSON payload, the latter specifies all the fields required to add a new asset. Finally, basing on the response to this call, the server sends a confirmation to the client or an error message. This can occur if the user specified wrong parameters, for example the name of a dealer that does not exist, or an identifier already present.

Ideally, this server could be controlled by an organization that needs to send products continuously. An example can be an e-commerce company, but also a production center of a particular raw material.

5.2 Dealer server

The dealer servers can be numerous. In a real implementation of the network there should be one for each organization offering the delivering service. This type of server is different from the others because it does not offer a web interface. Reminding the flow of the service, it only receives messages from the trackers and, when needed, sends confirmation transactions to the blockchain. Its logic is not trivial: the various messages received need to be handled smartly to avoid false negatives, but also not to miss any true positive.

This server can be split into two parts: the one that receives messages from the tracker and the one that processes them.

5.2.1 MQTT

All the communications between the trackers and the servers are based on MQTT. MQTT (Message Queuing Telemetry Transport) is an ISO standard messaging protocol. It is based on a publish-subscribe pattern: clients communicate with a server (called *broker*) as publishers or subscribers. Information is organized in topics, a hierarchical structure, similar to folders and files in a file system, using the forward slash as a delimiter. When a publisher sends a new message to the broker, at a certain topic, the broker then distributes the message to all the clients who have subscribed for that topic.

We decided to prefer MQTT over HTTP because it is particularly appropriate for IoT development. In fact, the former is a data-centric light weight protocol, while the latter is more document-centric and includes a significant overhead. MQTT is preferable when there are hard requirements of low battery and bandwidth usage, hence it is the best solution for our use case. A complete comparison between the two is reported in [24].

In our project, the server is the broker itself, while the devices are the publishers. No subscribers are needed, unless we want to scale in a particular way the servers, (e.g. adding other machines, which in that case would have to handle the messages as well). Every tracker communicates at its own topic (`/deviceIdentifier`), therefore the broker knows immediately which was the sender and can distinguish messages from different devices. Once received, it needs to decide whether to send a confirmation to the blockchain or not.

5.2.2 Messages management

Obviously, messages cannot contain any information about the goods being delivered. In fact, the IoT trackers do not have access to this data and can only send updates with the location and the nearby beacons. Consequently, a lot of messages are generated, but only a few of them need to be forwarded to the blockchain.

The server needs to periodically query the ledger to retrieve the list of goods to be delivered by its organization. Then, ideally, there should be a re-ordering phase of these deliveries, integrated with the company's internal management system, to assign each one to a single dealer. In this way, the server should have a unequivocal list of who will deliver what. Let us make this clear with an example.

The company X is a logistic company, its fleet is composed by four vehicles (X_1, X_2, X_3, X_4) and each one is equipped with the IoT tracker. X is registered as organization in our blockchain network. One day it queries the blockchain and it discovers that it has been chosen to take care of a delivery from the intermediate points A and B. X should already know B, in particular, the identifier of its beacon; if not, it can retrieve this data from the blockchain. Then, the company chooses one specific dealer, driving one specific vehicle, that will perform the delivery. In an internal register, the server should keep track that the vehicle X_i, carrying the device identified as i, is delivering a certain good to the intermediate point B, whose beacon identifier is b. When the dealer arrives at B, the tracker will detect the beacon and send an MQTT message containing b, at the topic `/i`. At that point, the server has to check that the beacon identifier corresponds to the one saved in the internal database and, if so, submit to the blockchain a transaction in which it confirms the delivery from the dealer. This complicated infrastructure is necessary because we want to avoid the case, although remote, of another dealer X_j "accidentally" sending a confirmation, without actually being the one assigned for the delivery.

5.3 Intermediate point server

This type of server is the other one directly involved in exchanges. The manager of the intermediate point should authenticate in the system and fill a form to confirm the reception of a certain good. In particular, this server has an internal database (we used MongoDB, but there are many alternatives) to keep the credentials of the users: in fact, there can be different employees working in shifts.

Practically, after logging in, the user can insert the identifier of a good (we assumed that it is directly readable from the product itself, maybe on a sticker); then, the server generates a transaction towards the blockchain to confirm that delivery. Clearly, if the identifier does not exist, the transaction will not be approved and an error message will be returned. It is important that the dealer controls

the correctness of this action, to defend themselves from a malicious employee at the exchange point. However, Fabric transactions are approved quickly enough to make this process smooth.

It is worth noting that the beacon, the other component involved on the exchange point side, is independent from the server. The goal of the two, in fact, is different: the beacon protects the dealer, because the tracker detects it and sends a proof of handover; on the other hand, the web interface protects the intermediate point operator, as it allows their confirmation. Keeping them separate enhances the security of the process. The are only linked in the blockchain ledger, because the beacon identifier must be traceable back to its own organization.

This approach is actually quite simple, too. There is no real need to link the server to the beacons. Unlike the trackers, beacons do not need a specific operating system nor a back end service. They are just hardware transmitters, which constantly broadcast their identifier using BLE (Bluetooth Low Energy). Once the UUID has been registered they do not require any further attention. However, if they are battery powered, it is important to recharge them or replace the battery periodically. This operation is not necessary with a significant frequency, as the battery life for most devices is approximately one year.

How servers of this type are deployed depends on which type of network we want to build. One possibility is following the same approach as the dealers', grouping the various intermediate stops in organizations that manage them. However, this is not a fixed aspect and the architecture supports different configurations.

5.4 Receiver server

The last server offers an interface to the receiver of the good. They can consult it whenever they want to check the updates of the delivery and it displays all the transactions that have occurred.

The logic is similar to the intermediate server one: the user inserts the good's identifier and the interface shows the related data, or an error for an invalid request. This information is retrieved from the blockchain using a query. In this service, though, there is no login in needed: we assume that the receiver has obtained the delivery identifier from the sender through another canal (which could be trivially by e-mail, as it happens with traditional services).

This service can be controlled by a single organization, using the ledger in read-only mode to support the receivers and offer them the information they need.

6 Testing and future developments

In the previous chapters we have explained how we modelled our solution. The result is an abstract architecture that still presents some points that need to be defined. To do so, some real requirements, coming from a client, need to be included. As it has been presented, the project is only a "proof-of-concept", that can represent a good base for an extended version adapted to a more specific use case.

Spindox usually builds custom services, hence the team was not interested in developing an entire platform to be used to ensure certain properties (such as, for example, the payment processor "PayPal"). Instead, the company wanted to explore the blockchain environment and understand, through the elaboration of this project, what are its benefits and drawbacks.

Our team only tested it in a local environment. Hyperledger offers wide support with its tools, that allowed us to test the network concretely simulating a deployed version. In particular, Composer provides a simple but still complete interface to work with the REST API available and develop the other elements on top of them. Passing from Composer to real Fabric, however, is not trivial and requires a thorough configuration of the various machines involved. The peers need to be deployed with respect to the specific use case: how to scale them correctly depends on the final requirements. In any case, we have given in Section 4.3 some general insights that need to be followed to guarantee a well distributed system. Also the consensus protocol has not been chosen yet: Hyperledger provides some good built-in alternatives and we mentioned an external one too. Concerning the latter, though, the integration might be quite complicated.

6.1 Limits

Hyperledger is a valuable solution that includes not only the properties of the blockchain, but also the ease of use and extensibility necessary to build a private network. The framework allowed us to build a system operating effectively, with the integration of the various interfaces and the IoT devices. However, the presence of an immutable ledger has negatives too. The trackers may generate a high number of messages, some of them unnecessary, and the server, managing them, will occasionally send false positive transactions. This is not, as a matter of principle, a problem: if the dealer tries to complete the delivery, but for some reason they do not succeed, this should be reported. Anyway, if a confirmation is sent only on the dealer's side, the system will be in an unclear state. A final receiver who checks the updates exactly in that moment might worry for no real reason. To solve this, we could implement some sort of "garbage collector" for these transactions, but it is impossible by design to delete entries from the blockchain. As a result, the only countermeasure is to program in the best way possible the servers and the firmware of the devices, to reduce this issue at the minimum.

A shared ledger containing all the transactions implies a lack of privacy, as well. As mentioned in Section 4.3, channels are a good measure to ensure privacy within a defined subset of members. Also precise access control rules (see Section 4.4.1) can help achieving this goal. Nonetheless, not always they are the best solution. Fabric guarantees strong authentication to all messages, but to obtain confidentiality too, the organizations need to perform some workarounds, since, by default, the ledger is plain. A centralized approach, from this point of view, would satisfy much more easily this requirement, by implementing private communications between the organizations and the central authority.

Also considering the framework itself, Hyperledger has some shortcomings. For instance, the fact that smart contracts can be written in a general purpose language is a potential risk, since the security that a DSL would provide is not guaranteed. It can be argued that, since an error in such programs can cost millions of dollars, the language should be safe by design. In addition, writing code for Hyperledger is not immediate and user-friendly: even the classic "Hello world" program can require

a significant number of rows. As recognized by both industry and academia, more code means more bugs.

All the communications using HTTP can be secured by including a TLS layer. Organizations in Fabric have their CAs by default, set up together with the network, while public accessible interfaces should implement certificates as well. On the other hand, MQTT is, by default, a vulnerable protocol, since messages are sent in clear. Unfortunately, the IoT hype makes developers often forget about security, but there are some measures that can be applied to guarantee a secure channel [21]. In every case, though, some overhead is included and inevitably this worsens the MQTT performances.

6.2 Extensions

The current solution has surely some extensions that can be included. In this section we point out a couple of them.

First of all, we did not include any additional use of the IoT trackers' updates. Since the servers receive continuously the dealers' location, the companies could build an internal system for vehicles tracking. Alternatively, an extension of the blockchain network logic could add transactions that update the assets with their current location, not only at exchange points. This adds no special value, compared to already existing solutions: the receiver would have to trust a single organization (the dealer's), exactly as happens nowadays with the tracking platforms exposed by almost every logistic company. Nonetheless, it can still represent a remarkable feature.

Another aspect that could be handled more smoothly is the other type of handover occurring in our use case: with this version only the "arrival" of a good at an intermediate point is tracked, while the confirmations of exchange for the following part of the route are not registered. Referencing Figure 2.1, for example, only the handover from D_1 to the employee in I_1 is written on the blockchain, hence when G is being delivered by D_2 towards I_2 the final recipient has not been informed. Such extension could be implemented by enabling an additional operation from the interface offered to intermediate points, but it should not be open to any attack from a malicious employee.

7 Conclusion

In this work we tried to introduce a reliable and secure model to solve the problem of logistics tracking, within entities that do not trust each other. This scenario can occur several times, in many different use cases. The participants can be couriers and simple employees working at exchange points, or even naval authorities and cargo ships.

We proposed a universal solution, which has not been developed under definitive requirements. It presents a precise idea and some defined mechanisms, but it needs to be adapted to a specific use case.

The problem has been firstly formalized, in order to identify the different entities involved in the scenario. This was necessary to have a defined and formal structure as a base. Then we presented the solution and we scoped down to its components.

In the first part we analyzed the state-of-the-art and the market of IoT trackers, in order to find a device for the geolocation of the dealers. We wrote the firmware and defined a protocol for the communication of updates. Then we moved to the architecture: the goal was to decentralize the solution and create a network of collaborating organizations. After introducing the blockchain and its characteristics, we decided to use a permissioned network. We described how the technology works and its main features. Finally, we reported how we programmed the various servers involved in the architecture.

The solution solves the problem with respect to the initial requirements. Exchanges are registered permanently on the distributed ledger, which can be consulted by the final recipient. Every entity involved in a particular delivery has to give its confirmation, thus each one is always protected against malicious counterparts. These confirmations are done automatically (for the dealers) or manually (for the intermediate points). However, the proposed model has some negatives, primarily related to the use a blockchain, but also for the necessity of a contribution from the people directly involved.

In fact, as explained in the thesis, the blockchain has shown that all the hype that surrounds it is not completely justified: many times this technology is not necessary and in any case it brings a number of limits in terms of performance and maintenance. Nonetheless, permissioned blockchains have the potential to decentralize some services effectively, if well implemented.

The other weak point of our solution is that it requires a human action to function properly: it is not fully automated and this inevitably makes it less secure. Nevertheless, unlike the traditional approach (based on signatures), the model we propose guarantees complete reliability if properly respected, whereas the other one is vulnerable to counterfeiting attacks.

Bibliography

[1] Accent systems. https://accent-systems.com/.

[2] Docker. https://www.docker.com/.

[3] Estimote. https://estimote.com/.

[4] Hyperledger. https://www.hyperledger.org/.

[5] Kafka. https://kafka.apache.org/.

[6] Micropython. https://micropython.org/.

[7] Micropython libraries. https://github.com/micropython/micropython-lib.

[8] Pycom. https://pycom.io/.

[9] Pycom libraries. https://github.com/pycom/pycom-libraries.

[10] Raft. https://raft.github.io/.

[11] Sensolus. https://www.sensolus.com/.

[12] F. Adelantado, X. Vilajosana, P. Tuset-Peiro, B. Martinez, J. Melia-Segui, and T. Watteyne. Understanding the limits of LoRaWAN. *IEEE Communications magazine*, 55(9):34–40, 2017.

[13] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[14] V. Buterin. Ethereum: a next generation smart contract and decentralized application platform. 2013.

[15] C. Cachin and M. Vukolić. Blockchain consensus protocols in the wild. *CoRR*, abs/1707.01873, 2017.

[16] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[17] P. García López, A. Montresor, and A. Datta. Please, do not decentralize the internet with (permissionless) blockchains! *CoRR*, abs/1904.13093, 2019.

[18] K. Mekki, E. Bajic, F. Chaxel, and F. Meyer. A comparative study of LPWAN technologies for large-scale IoT deployment. *ICT Express*, 5(1):1–7, 2019.

[19] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[20] M.E. Peck. Do you need a blockchain? This chart will tell you if the technology can solve your problem. *IEEE Spectrum*, 54(10):38–60, 2017.

[21] M. Singh, M.A. Rajan, V.L. Shivraj, and P. Balamuralidhar. Secure MQTT for Internet of Things (IoT). In *2015 Fifth International Conference on Communication Systems and Network Technologies*, pages 746–751. IEEE, 2015.

[22] J. Sousa, A. Bessani, and M. Vukolić. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 51–58. IEEE, 2018.

[23] Y.P.E. Wang, X. Lin, A. Adhikary, A. Grövlen, Y. Sui, Y. Blankenship, J. Bergman, and H.S. Razaghi. A primer on 3GPP narrowband Internet of Things (NB-IoT). *CoRR*, abs/1606.04171, 2016.

[24] T. Yokotani and Y. Sasaki. Comparison with HTTP and MQTT on required network resources for IoT. In *2016 international conference on control, electronics, renewable energy and communications (ICCEREC)*, pages 1–6. IEEE, 2016.