Secure DHT with Blockchain technology

Nuno Filipe Mateus Nogueira

Thesis to obtain the Master of Science Degree

Telecommunications and Informatics Engineering

Examination Committee

Presidente: Nome do Presidente
Orientador: Nome do Orientador
Co-orientador: Nome do Co-orientador
Vogais: Nome do Vogal 1
Nome do Vogal 2

Mês e Ano

Acknowledgments

A few words about the university, financial support, research advisor, dissertation readers, faculty or other professors, lab mates, other friends and family...

Abstract

The abstract describes the objective, the content of the work and its conclusion. Use a maximum of around 250 words.

Keywords: keyword1, keyword2,...

Resumo

O resumo analítico descreve o objectivo, o conteúdo do trabalho e as conclusões, também designado por resumo ou abstract, deve ser escrito em português e inglês, com um máximo de 250 palavras cada.

Palavras-chave: palavra-chave1, palavra-chave2,...

Contents

Acknowledgments iii

Abstract v

Resumo vii

List of Figures xi

List of Tables xiii

Acronyms xv

1 Introduction 1

- 1.1 Background 2
  - 1.1.1 Global Registry 2
  - 1.1.2 Domain Registry 2
  - 1.1.3 Local Registry 2
- 1.2 Proposed Solution 2
- 1.3 Thesis Contribution 3
- 1.4 Outline 3

2 Related Work 5

- 2.1 P2P Networks 5
  - 2.1.1 Unstructured 5
  - 2.1.2 Structured 5
  - 2.1.3 Kademlia 6
- 2.2 Distributed Hash Table (DHT) security considerations 7
  - 2.2.1 Routing Attacks 7
  - 2.2.2 Storage and Retrieval Attacks 7
  - 2.2.3 Sybil Attack 8
  - 2.2.4 Eclipse Attack 10
- 2.3 Public Key Infrastructure 11
  - 2.3.1 Certificate Authorities 13

2.3.2 Web Of Trust 16
2.4 Blockchain and the Bitcoin protocol 17
2.4.1 Block structure 17
2.4.2 Merkle Trees 18
2.4.3 Proof of work 19
2.4.4 Blockchain applications 19
2.4.5 Smart Contracts 20
2.4.6 Ethereum 20

3 Architecture **23**
3.1 Requirements 23
3.2 Overview 23
3.3 Distributed Hash Table 24
3.4 Decentralized Public Key Infrastructure 25
3.4.1 IDChain Smart Contract 25
3.4.2 IDChain Application Programming Interface (API) 29
3.4.3 IDChain Management Application 30
3.5 Overlay network processes 31
3.5.1 Node bootstrap and registration 31
3.5.2 Node message routing 31
3.5.3 Eclipse and Sybil attacks defense 32
3.5.4 Data replication 32
3.6 Comparison with a CA 33

4 Implementation **34**
4.1 Implementation Options 34
4.2 Architecture 34

5 Evaluation **36**
5.1 Tests Objectives 36
5.2 Tests Scenarios 36
5.3 Test Results 36

6 Conclusions **37**
6.1 Summary 37
6.2 Achievements 37
6.3 Future Work 37

A Vector calculus **39**
A.1 Vector identities 39

Bibliography **43**

List of Figures

2.1 X.509 version 3 certificate structure . . . 13
2.2 X.509 v2 CRL structure . . . 15
2.3 Blockchain structure and blocks content. . . . 18
2.4 Merkle tree example . . . 19
2.5 Simple data storage contract in Solidity language. . . . 21

3.1 Overview of solution architecture. . . . 24
3.2 Smart contract *Certificate* structure fields. . . . 26
3.3 Web of trust mechanism. . . . 27
3.4 Smart contract *Entity* structure fields. . . . 28
3.5 In Scenario A bootstraper entities *A, B and C* vouch for entity *D*, which vouches for entity *E* in conjunction with entities *A* and *C*. In Scenario B, entity *C* unvouches entity *D* which is considered invalid and therefore also entity *E* is considered invalid. . . . 29
3.6 . . . 30
3.7 Additional steps added in TLS handshake verification. . . . 32

List of Tables

3.1 IDChain API specification 31
5.1 Table caption shown in TOC 36

List of Acronyms

OTT Over-the-top

Hyperties Hyperlinked Entities

M2M Machine-to-Machine

IoT Internet of Things

DHT Distributed Hash Table

GUID Global User Identifier

SP Service Provider

DPKI Decentralized Public Key Infrastructure

PKI Public Key Infrastructure

P2P Peer-to-Peer

RTT Round-Trip Time

MAC Media Access Control

CA Certificate Authority

HTTPS Hyper Text Transfer Protocol Secure

SSH Secure Shell

TLS Transport Layer Security

SSL Secure Sockets Layer

CRL Certificate Revocation List

OCSP Online Certificate Status Protocol

PGP Pretty Good Privacy

EVM Ethereum Virtual Machine

DAO Decentralized Autonomous Organization

CSR Certificate Signing Request

IPFS InterPlanetary File System

API Application Programming Interface

SPA Single Page Application

Chapter 1

Introduction

Nowadays with the current Internet infrastructure and the provided services built on top of it, the old telecommunications operators based services, like voice telephony, are losing importance.

Over-the-top (OTT) players, like Google and Skype, are dominating the communications market with a no additional cost and closed ecosystem solutions. New users will choose to use the services, that are used by the majority of their social environment.

OTT services, by working in the closed ecosystem don't need to work in interoperability between services and communications standards, allowing them to be more competitive, agile and lead communication and multimedia innovation. This could be problematic since it causes vendor lock-in and limits the portability of user identity and data, hinders innovation and block new entrants.

On the other hand, we have the worldwide Telco ecosystem that provide an highly reliable service and strong trustful identity. Since is necessary to achieve worldwide service interoperability, the services provided rely on well-defined standards. These standards need to be agreed upon and defined, increasing the time to market of potential new services. Telcos are also geographically restricted, so the deployment of new worldwide services could not be possible without roaming agreements in-place, which severely restricts Telcos in driving innovation.

The reTHINK [1] project goal is to design a new peer-to-peer network infrastructure for communications based on Web technologies, that allow dynamic trusted relationships between distributed apps and a portable identity model, leveraging the advantages of the federated Telco and OTT model.

This dynamic trusted relationship is created by using Hyperlinked Entities (Hyperties), a web microservice paradigm, that enable the execution of trustful services in a web environment on user devices or network servers. In order to achieve interoperability, the communication between hyperties is based on the *Protocol-On-The-Fly* [1] concept, that allows using standard network protocols through a common API enabling communication between different hyperties from different service providers. The hyperty concept allows to extend the communications beyond normal telephony and messaging, where even services using Machine-to-Machine (M2M) and Internet of Things (IoT) systems could be built.

---

[1] https://rethink-project.eu/

1.1 Background

One of the main components in the reTHINK architecture is the **Registry** service. The Registry service is a key-value based directory service, that facilitates the management and lookup of hyperty instances running in users devices.

The Registry service must have the following requirements:

- *Provide fast query response time*, since it will be accessed when establishing communication;
- *Scalable*, since it will be a worldwide deployed service;
- *High availability*, is necessary for communication establishment;
- *Data consistency*, the hyperties information must be always up up to date, so it is possible to start the communication setup.

The Registry service is sub-divided in three components: *Global Registry*, *Domain Registry* and *Local Registry*.

1.1.1 Global Registry

The Global Registry is a key-value store based on the DHT technology, that stores user identifiers in hyperty services, index by a Global User Identifier (GUID).

1.1.2 Domain Registry

The Domain Registry are run by Service Provider (SP) and allows users to lookup users hyperty instances using user hyperty service identifier. It uses a client-server model, which allows to handle a high data update rate.

1.1.3 Local Registry

Component in the device runtime that manages hyperty instances running in the runtime and contact the Global and Domain Registry.

1.2 Proposed Solution

The major goal of this thesis is to ensure the security of the DHT necessary to build the Global Registry system.

In order to achieve this, a set of problems need to be solved:

- **Mitigate common DHT attacks**- Several different type of attacks targetting the DHT exist. A in-depth analysis of each one of them and existing solutions is necessary;

- **Ensure routing messages integrity**- it is necessary to secure the routing scheme of the DHT by using public-key cryptography;
- **Users and DHT nodes public key distribution**- Since routing messages should be encrypted, it will be necessary that nodes be able to distribute each other keys;
- **Deal with key compromise**- allow nodes compromised public keys to be revoked;
- **Maintain system decentralized**- solve all the aforementioned problems, maintaining a decentralized architecture.

The solution to these problems will consist in building the DHT in conjunction with a simple Decentralized Public Key Infrastructure (DPKI), using blockchain technology. The blockchain will ensure the DPKI is totally decentralized and achieve data integrity and prevent tampering of its contents. In order to prevent some of the most common DHT attacks, the blockchain cryptocurrency will be used so it is possible to guarantee that an attacker will incur in high costs to launch attacks against the DHT.

1.3 Thesis Contribution

The following list present the expected contributions of this thesis:

- **DHT Security** - Design a DHT prototype with improved resiliency against common attacks;
- **Public Key Infrastructure (PKI)** - Design a simple PKI prototype that will run in a decentralized system and run together with a DHT;
- **Blockchain based applications** - Show that is feasible to build common distributed applications in a decentralized manner leveraging blockchain technology;
- **System performance and evaluation** - assess the performance cost of the proposed system.

1.4 Outline

This document describes the research and work developed and it is organized as follows:

- **Chapter1** presents the motivation, background and proposed solution.
- **Chapter2** describes the state of the art of the technologies used in the solution architecture.
- **Chapter3** describes the system requirements and the architecture of the solution.
- **Chapter4** describes the implementation of the solution and the technologies chosen.
- **Chapter5** describes the evaluation tests performed and the corresponding results.
- **Chapter6** summarizes the work developed and future work.

Chapter 2

Related Work

This section address the state of the art of the research topics relevant to our proposed work: P2P Networks, DHT security considerations, Public Key Infrastructure, Blockchain and the Bitcoin protocol.

2.1 P2P Networks

Peer-to-Peer (P2P) networks is a distributed systems architecture where equal and autonomous entities (peers) are interconnected and form a network with the objective of sharing distributed resources. The P2P network model allows peers to self-organize into a network topology, that is able to deal with failures and adapt the network topology with a variable rate of joining and exiting nodes (churn rate). [2] P2P systems networks are usually categorized as *Structured* or *Unstructured* [3].

2.1.1 Unstructured

In an Unstructured overlay network, nodes join the network by connecting to other nodes and without having a prior knowledge of the topology, therefore creating a random network structure. Peers search for content using flooding as a mechanism to query for the other nodes content that match the query. Even though the flooding mechanism is resilient for locating highly replicated content and is resilient to a high churn rate, it is not suitable to locate rare content and it does not scale well for a high number of nodes, due to the high load generated by the queries. P2P systems like Gnutella[4] use an unstructured overlay network.

2.1.2 Structured

Structured overlay networks use node identifiers to organize the network and maintain the overlay network topology when new nodes join or exit the network. The content is placed in specific locations, therefore allowing more efficient queries. The most common structured overlay pattern is the Distributed Hash Table. DHT systems assign random node identifiers to peers from a large identifier space. Datasets also are assigned unique identifiers from the same identifier space, called *keys*, by applying a cryptographic hash function to the data. This allow to create an index, where a key identify the position of the corresponding dataset in the network, therefore making possible to retrieve the data from a live peer in the network. Peers maintain routing tables that store the neighbor peers (identifier and IP address) in the identifier space, that are necessary to forward routing messages across the overlay network until it reaches the destination node. Several DHT-based solutions exist, which implement different overlay network structure and routing schemes.

Chord [5] and Pastry [6] organize nodes in a circular identifier space, where each node is responsible for a section of the circular identifier space and for forwarding routing and lookup messages along the neighbor nodes, until it reaches the node responsible for the given key. Kademlia [7] map nodes into a balanced binary tree recurring to XOR operation as a distance metric to perform parallel lookups. A more comprehensive description of Kademlia is given in the next sub-section.

2.1.3 Kademlia

In Kademlia a 160-bit uni-dimensional address space is used for node and key identifiers, which could be represented as a balanced binary tree. The key-value pairs are stored in the node closest to the key, according to the distance metric. Routing in Kademlia is based on the XOR distance metric, $d(a, b) = a \oplus b$, for the notion of distance between two identifiers. This metric as two useful properties:

- Is **unidirectional**, which means that for a given distance there is only one identifier at that distance, consequently, all lookups for the same key converge along the same path, so is possible to do *caching* of keys;
- Is **symmetric**, which allows the node to update the routing table with the received search messages.

Messages may be sent to any node of an interval, making it possible to send messages in parallel, reaching first the node with smaller latency. In order to route query messages, nodes need to keep a list of *k* nodes for each sub-tree where it is not present. This lists are called *k-buckets* and are sorted by time last seen, i.e time elapsed since last message. k-buckets have a size *k*, where *k* is choosen such that is very unlikely that all nodes fail within an hour.

Every message received by a node, triggers a update in the k-bucket for the sender identifier. If there is available space in the k-bucket, the new identifier is inserted at the tail of the k-bucket, or, if already present, moved to the tail. If the identifier is new and the bucket is full, the oldest element is pinged. If the node responds it's moved to the tail of the list and the new one is discarded. Otherwise, it's removed and the new node is placed at the tail. This provide resistance to denial of service attacks, because as the routing table favors old nodes, it is not possible to flush them by flooding the DHT with new nodes.

Kademlia needs to maintain a basic routing table by using k-buckets, and maintaining information for each sub-tree, knowing at least one node for each of the sub-trees. When a new node *b* join and contacts a node *c*, *b* adds *c* to a k-bucket (if full the k-bucket is split between the two) and performs a lookup for himself. Then *b* refreshes k-buckets further away than closest neighbor, therefore populating its own k-buckets and inserting himself into other k-buckets nodes. Since key-value pairs may expire and nodes with stored values may leave, Kademlia is a *soft state* system, and therefore need data republishing.

Data republishing is done by republishing key-value pairs every hour.

2.2 DHT security considerations

There are several security considerations [8] to take into account when building a DHT. They could be summarized into four categories:

- *Routing Attacks*
- *Storage and Retrieval Attacks*
- *Sybil Attack*
- *Eclipse Attack*

2.2.1 Routing Attacks

Routing requests is an essential component in any DHT, so it is critical that routing tables are correct in DHT nodes. Since each node has to maintain his own routing tables and update them accordingly, there are multiple attack vectors that an attacker can take advantage of [9]:

Incorrect Lookup Routing. A malicious node could forward lookups to an incorrect or non-existent node.

Incorrect Routing Updates. Since each node builds its routing table using information from other nodes, a malicious node could corrupt the routing tables of other nodes by sending incorrect updates.

Network Partition. In order to bootstrap into the DHT network, a node must contact some participating node. This makes the node vulnerable to entering an incorrect network. The first node to be contacted may redirect the new node to a different partition, under malicious control.

2.2.2 Storage and Retrieval Attacks

A malicious node may be able to attack the underlying storage layer of the DHT. For example, it might claim to store data when asked, but then refuse to serve it to clients. Dealing with this attack requires the use of data replication, but it must be handled so that no single node is responsible for replication or facilitating access to the replicas, i.e avoid single points of responsibility. Clients should be able to determine the correct nodes to contact for replicas and obtain the data from these replicas, as a way to prevent single points of responsibility.

Kademlia prevents this sort of attack, because it does parallel searches and all the searches for a given identifier converge on the same path. Combined with data republishing and replication, the client may contact several nodes to ensure the data correctness.

2.2.3 Sybil Attack

The Sybil Attack is an attack that exploits a distributed system when it fails to guarantee that distinct identities refer to distinct entities [10]. In a P2P system - like a DHT - if an attacker controls a fraction of the node identifiers, it is possible to create a collusion of malicious nodes in the DHT and even pollute the routing tables of honest nodes. A Sybil Attack amplifies the effect and reach of other attacks, such as the ones described earlier. The Sybil Attack is a real threat to any overlay network, and there is enough evidence that this attack is possible in real deployed networks, as BitTorrent Mainline DHT[11].

The Sybil Attack defense mechanisms can be sub-divided in the following categories:

Centralized certification

Douceur argues that the only way to have a unique direct relation between an identifier and an entity is by having a central trusted authority [10].

One proposed solution is to use certified node identifiers[9] issued by a trusted **ca!** (**ca!**) that assigns node identifiers and signs node identifier certificates. Each certificate binds a node identifier to a public key and an IP address. The inclusion of the IP address prevents an attacker from moving the certificates across several nodes, minimizing this way the number of attacker nodes. In order to avoid an attacker from obtaining several node identifier certificates, these certificate must be bought from the **ca!** entity.

Even thought this solution allows control over who joins the network, it has the disadvantage of requiring a trusted centralized entity to manage the node identifier attribution. In our view, the cost of this solution (cost of certificates and inability to change IP address) makes it unfeasible for most public DHTs.

Network characteristics

One proposed solution that uses network characteristics is the *net-print*[12] mechanism. The *net-print* are the node physical characteristics: node default router IP address, MAC address and vector of Round-Trip Time (RTT) measurements between the node and a set of routers (landmarks). The net-print data can be easily verified by other nodes, by directly measuring the net-print of the node and comparing it with the claimed net-print. This prevents possible sybil attacks by malicious nodes. A problem in this mechanism are variable network conditions, that may cause the identity verification to fail, since the net-print values reported and the directly measured values would be different. A tolerance in value deviation could be a possible solution, but this could diminish the security efficiency of the system.

Bazzi et al.[13] also proposed a sybil defense mechanism that leverages network characteristics to create network coordinates. The proposed solution tries to solve the *group distinctness problem*, i.e determining the number of distinct entities on which a group of identities reside. By being able to be assured that, for example, in two separate groups of identities for any two identities chosen from different groups they are distinct, this allows to achieve redundancy in the execution of a remote operation by sending the operation to the two groups. Each entity is located at a point in the geometric space (*d*-dimensional Euclidean space or a sphere), therefore the transmission time of a message between two points A and B gives a upper-bound on the distance *d(A,B)* between the points. This solution takes into account the following assumptions:

- the distance between two points satisfy the metric properties of a symmetry and triangle inequality;
- the distance between a pair of points is a non-decreasing function of roundtrip delay between them.

The system model considers a set of *beacons* and a set of *applicants*, which all together constitute the set of *participants*.

This system comprehends two essential elements: **geometric certificates** and **group distinctness test**.

A *geometric certificate* is a set of signed distance values between the beacons and the applicants, calculated by applicants and beacons responding to probe messages. The *group distinctness test*, for example, could be a *two-distinctness test* represented as a function $D : C \times C \rightarrow \{true, unknown\}$, where *C* represents a geometric certificate. If by applying $(c1, c2)$ to the function the result is *true*, then the entities with these geometric certificates are distinct.

The main advantage of this system, is that it provides a good basis for a redundancy protocol, by guarantee that identities in different groups are not controlled by the same identity. The major problem with this system is that it can be easily circumvented, if an attacker use several distributed nodes, that will be assigned to different groups. Also the system is tested on a network that could not exhibit the same delay characteristics as of the Internet.

Computational puzzles

S\Kademlia[14] proposes a *dynamic crypto puzzle* that attaches a barrier to the generation of several node identifiers. After the node identifier is generated from hashing the user public key, a random number *N* is choosen and the value of *P* is calculated accordingly to the function $P := Hash(NodeID \oplus N)$, where $\oplus$ is the XOR operation. This function is repeatedly calculated by changing the value of *N*, until there is a value of *P* that is preceded by *c* zero bits, where *c* is a constant that can be adjusted to increase or reduce the difficulty of the crypto puzzle. The verification of the crypto puzzle is done every time a node receives a signed message.

Computational puzzles is a decentralized solution that can effectively limit the number of Sybil identities. But it has the disadvantage that it is a solution that can only mitigate the attack and not impede it, and forces honest node to spend computing resources to solve the puzzles.

Several other solutions exist that use *social networks*[15][16] or *game theory*[17]. These solutions require an *a priori* social network or social relationships between participants or, in the case of game theory, the use of economic incentives through the implementation of a currency, turning these solutions infeasible in the scope of this project.

2.2.4 Eclipse Attack

In an Eclipse Attack[18] malicious nodes collude with the objective of fooling other nodes into adding them to their neighbor set, poisoning their routing table. If successful, the attacker can be ensured that all messages from that node to the overlay network and vice-versa are routed through at least one malicious node. This gives the malicious node the ability to block and inject messages, providing an incorrect view of the overlay network to the honest nodes or even drive a denial of service attack. It is possible to assume that an Eclipse attack is closely related to a Sybil attack, but even a small number of malicious nodes with different identities could be able to produce an Eclipse attack. A specific security mechanism is required to prevent this sort of attack. In order to impede the attack some of the main requirements are:

- the nodes be unable to choose their node identifier, which protects against Eclipse attacks that try to target a specific node or region of the routing table;

- increase the difficulty of influence other nodes routing table, which increases the difficulty of an attacker block honest node's correct view of the overlay network.

The previously presented *Routing Attacks* are closely related with Eclipse attacks, so the solutions presented in this section will also help mitigate those attacks.

Singh et al.[18] presented a solution that is based on the fact that the *in-degree* of malicious nodes is higher that the average in-degree of honest nodes. This way, is possible to prevent an Eclipse attack by choosing nodes with an *in-degree* below a certain threshold. But malicious nodes may connect to honest nodes in order to increase their in-degree, so is also necessary to bound the out-degree of the nodes. This is implemented by building an anonymous distributed auditing mechanism, where every node challenges anonymously, each member of the neighbor set for their *backpointer set*: a list of the nodes that contain the challenger node in their neighbor set. If the count of entries in backpointer set is bigger that the in-degree bound, or the challenger node doesn't appear in the backpointer set, the challenged node is removed from the challenger node neighbor set. A similar procedure is done to check the out-degree bound, by verifying if the neighbor set size of a node's backpointer set members is below the out-degree threshold. In order for this system to work, the challenger node must remain anonymous, or a malicious nodes could fake the response to a challenge, by creating a fake backpointer set with a size below the in-degree threshold and adding the auditing node. So, it is necessary relay the challenge through an *anonymizer node* to hide the challenger identity. Since this anonymizer node belongs to the network and could be malicious, several audits must sent through different anonymizers at random times and the audit response must be digitally signed. A node A that challenges node B, selects a node randomly from the l numerically closest nodes to the hash H(B). The expected fraction of malicious nodes in this subset is equal to the fraction of malicious nodes in the overlay network. Considering that in the network, every node is considered malicious if it answers less that k out of n challenges correctly, the following probabilities could be calculated:

- **Honest node is considered malicious:**

$$\sum_{i=0}^{k-1} \binom{n}{i} f^{n-i}(1-f)^{i} \tag{2.1}$$

- **Malicious node pass the audit undetected:**

$$\sum_{i=0}^{k-1} \binom{n}{i} [f+(1-f)c/r]^{i}[(1-f)(1-c)]^{n-i} \tag{2.2}$$

Where *f* is the fraction of malicious nodes in the overlay network, *c* is the probability of malicious nodes answering the challenge, and *r* is the ratio of the size of the true set (real backpointer set) versus the maximum allowed size. This calculation is needed, since a malicious node may adopt a strategy where he responds with only a subset of his backpointer set with a size equal to the maximum allowed size. The authors state, as an example, in a scenario that a node is considered malicious if he answers less than *k* out of *n* challenges, with $f \leq 0.25$, $n = 24$, $k = 12$ and $r \geq 1.2$ the false positive probability is around 0.2% and malicious nodes are detected with a probability of at least 95.9%.

The advantage of this system is that it leverages the overlay network primitives to build a simple and efficient algorithm to detect and prevent Eclipse attacks. But the system is only effective with a small degree threshold, which increases the lookup time when there are no attack taking place.

S\Kademlia proposes a static crypto puzzle to generate node identifiers that is based on repeatedly generating a key pair and double hashing the public key, until there is *c* preceding zero bits in the hash. This solution only deals with the node identifier generation, since it randomizes the process and don't allow the node to choose a node identifier freely. Therefore, this solution is only efficient in situations where an targeted Eclipse attack may occur. Used in conjuction with the aforementioned dynamic crypto puzzle it will also limit sybil attacks.

2.3 Public Key Infrastructure

Through encryption and digital signatures[19], public-key technology[20] provides:

- **Confidentiality**: the data must not be made available or disclosed to unauthorized users;
- **Non-repudiation**: is a property that ensures that if a user signed data, he cannot deny signing that data. This prevents a entity from denying having performed a specific action;
- **Authentication**: is the process of confirming the identity of a user, preferentially without sending secret information through the network;
- **Integrity**: is the property that guarantees that data has not been modified. By using digital signatures it is possible to check if the digitally signed data has been altered since it was signed.

A Public Key Infrastructure[21][22] is a system that provides public-key encryption and digital signature services. It is structured as a framework that consists of security policies, encryption mechanisms and applications, with the propose of generating, storing and managing keys and certificates.

Therefore a PKI must manage the whole lifecycle of each key pair[23]. This lifecycle comprehends several tasks necessary to provide a secure management of these keys, and can be divided in three different phases: *key generation*, *key usage* and *key invalidation*.

In the *key generator* phase, it is necessary to provide users with the ability to generate key pairs. These key pairs may be generated by the user, preventing the private keys from being disclosed to unauthorized parties, but this key generation may not be possible in computational restricted devices. So is possible for a PKI infrastructure to generate key pairs for the user, but in this case, as a third-party generates the keys, the user private keys will also be know by the third-party which could bring some security concerns.

The *key usage* phase is one of the most important phases in the key pair lifecycle, since it has the task of *making the public keys available to users.*: it is not only necessary to publish the keys, it is also necessary to verify their authenticity and validity. If an attacker is able to replace an user public keys with his own public keys, he could impersonate that user and therefore decrypt messages sent to the user or sign documents on behalf of the user. In the key usage phase, there is also a task of managing key backups. The use case of key backups is the ability to provide users a mechanism for key recovery for when they lose the data private decryption key and need to access the encrypted data. As in the *key generation* phase, for a PKI infrastructure to be able to provide this backup mechanism, it will be necessary to store the user's private keys, which could bring potential security issues.

The last phase in the key pairs lifecycle is the *key invalidation*. One of the main tasks of the PKI in this phase is dealing with public keys that became insecure, due to a broken cryptosystem or if stolen. Two courses of action are possible in this scenario: *destroy* or *archive* the key. In the case of invalid public keys, they can be deleted since they cannot be used again safely, but private keys may be archived, so it may be possible to still access data that was encrypted with the corresponding public key.

One of the fundamental elements of the PKI infrastructure are the certificates. Certificates provide an authenticity proof for public keys, by binding an entity to a specific public key, through the signature of a third-party. If the user requesting the public key trusts the third-party, verifying the digital signature of the certificate is sufficient to ensure the user of the authenticity of the public key.

Usually the structure of the certificate contains at least:

- **Name of the entitity**;
- **Public key** bound to the entity;
- **Cryptographic algorithm** used by the public key;
- **Certificate serial number**;
- **Certificate validity period**;
- **Issuer of the certificate**;
- **Restrictions on the usage of the public key**.

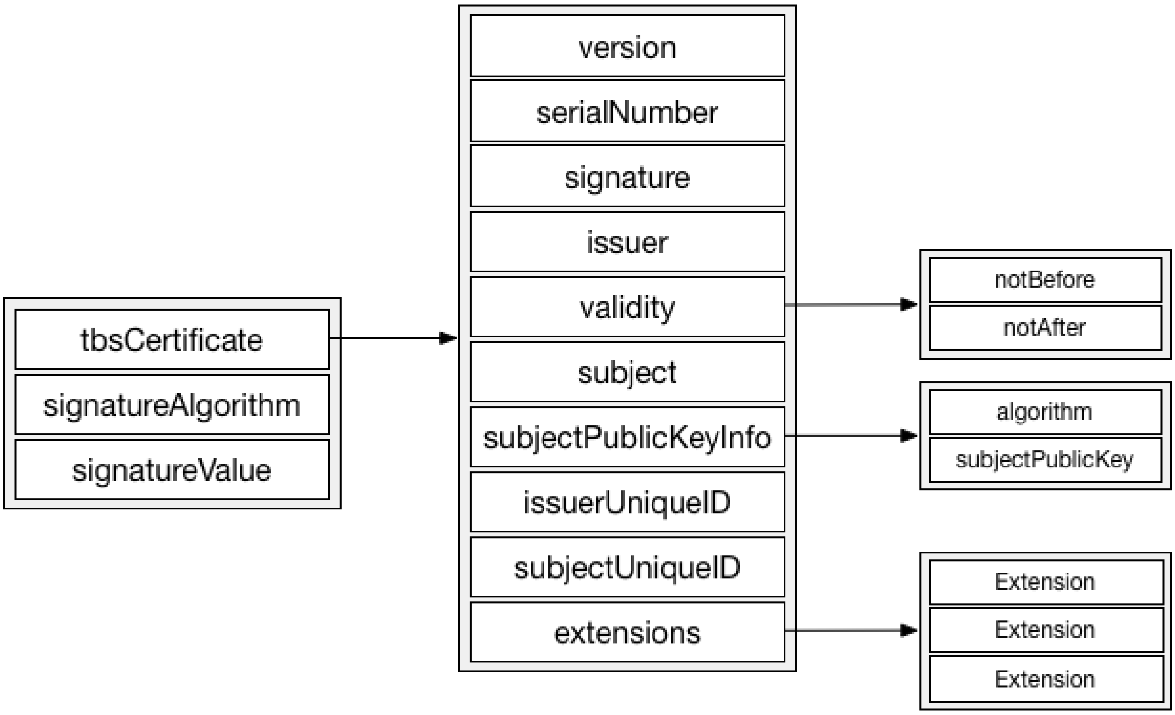

tbsCertificate
signatureAlgorithm
signatureValue
version
serialNumber
signature
issuer
validity
subject
subjectPublicKeyInfo
issuerUniqueID
subjectUniqueID
extensions
notBefore
notAfter
algorithm
subjectPublicKey
Extension
Extension
Extension

Figure 2.1: X.509 version 3 certificate structure

Several solutions that implement a PKI infrastructure exist: Certificate Authorities and Web of Trust.

2.3.1 Certificate Authorities

Certificate Authorities (CAs) are the main model of PKI used nowadays. In a CA model of PKI there is a third party that authenticates entities by issuing a digital certificate.

The X.509 standard[24][25] is one of the most used CA-based PKI infrastructures nowadays, and it is supported by several communication standards like HTTPS, SSH and TLS/SSL.

It comprehends the following components:

- **End entity**: users of PKI certificates or end user system that are the subject of a certificate;
- **Certification Authority**: authenticates entities in a transaction;
- **Registration Authority**: system responsible for the interactions between the end user and CA. It receives entity requests, processes and validates them, directs them to the CA for posterior processing and forwards the processed certificates to the user;
- **CRLs issuer**: generates and stores Certificate Revocation Lists (CRLs);
- **Repository**: stores and distributes CRLs and certificates to the end users.

The *X.509 certificate* contains several fields as depicted in Figure 2.1:

- **tbsCertificate**

version: describes the version of the encoded certificate;

serialNumber: unique positive integer assigned by the CA;

signature: describes the signature algorithm used by the CAto sign the certificate;

issuer: specifies the entity that signed and issued the certificate. Contains a ASN.1 string called *distinguished name* (DN) that describes a hierarchical name composed of several attributes such as country name, organization, etc;

validity: specifies the validity period of a certificate, and contains two date fields - *notBefore* and *notAfter* - that indicate, respectively, a point in time where the certificate was not valid yet and a point in time where the certificate is not valid anymore;

subject: describes the entity that owns the certificate and is associated with the public key stored in the *subjectPublicKeyInfo* field;

subjectPublicKeyInfo: this field contains two sub-fields - *algorithm*, used to identify the algorithm which the key uses, and *subjectPublicKey* which contains the subject public key;

issuerUniqueID/subjectUniqueID: unique identifier for the subject and issuer;

extensions: sequence of one or more certificate extensions that allow to add additional attributes associated with the certificate, providing support for additional PKI processes.

- **signatureAlgorithm**: contains the identifier of the cryptographic algorithm used to sign the certificate;
- **signatureValue**: contains a digital signature of the *tbsCertificate* content;

In X.509, the certificate revocation process supports two possible mechanisms:

- **Periodic Publication Mechanisms**;
- **Online Query Mechanisms**.

In the **Periodic Publication Mechanism** the process consists of periodically issuing a Certificate Revocation List. A CRL is a timestamped list signed by the CA or CRL issuer that specifies the revoked certificates.

In Figure 2.2 it is possible to view the several fields in the CRL definition:

- **tbsCertList**: contains the certificate list to be signed, which is represented by the following fields:

version: CRL version definition;

signature: identifier of the cryptographic algorithm used to sign the CRL;

issuer: describes the entity that signed and issued the CRL;

thisUpdate: issue date of the CRL;

nextUpdate: issue date of the next CRL. A new CRL must be issued before this date;

```
CertificateList  ::=  SEQUENCE  {
  tbsCertList          TBSCertList,
  signatureAlgorithm   AlgorithmIdentifier,
  signatureValue       BIT STRING  }

TBSCertList  ::=  SEQUENCE  {
  version                 Version OPTIONAL,
                          -- if present, MUST be v2
  signature               AlgorithmIdentifier,
  issuer                  Name,
  thisUpdate              Time,
  nextUpdate              Time OPTIONAL,
  revokedCertificates     SEQUENCE OF SEQUENCE  {
    userCertificate         CertificateSerialNumber,
    revocationDate          Time,
    crlEntryExtensions      Extensions OPTIONAL
                          -- if present, version MUST be v2
  }  OPTIONAL,
  crlExtensions           [0]  EXPLICIT Extensions OPTIONAL
                          -- if present, version MUST be v2
}
```

Figure 2.2: X.509 v2 CRL structure

revokedCertificates: the list of revoked certifies. Each revoked certificate is identified by its serial number in the *userCertificate* field and should also specify the *revocationDate* field;

crlExtensions: optional field, meant to add additional attributes in the CRL;

- **signatureAlgorithm**: contains the identifier of the cryptographic algorithm used to sign the *tbsCertList* content;
- **signatureValue**: contains a digital signature of the *tbsCertList* content;

In a *periodic publication mechanism*, one of the key aspects is the *revocation delay*, that is the delay between the report of a revocation and the publishing of the updated CRL in the repository. This is an important aspect, because there is the security risk of a user trusting an already revoked certificate. This turns out to be a limitation, since the update scheduling of the CRLs may not be known.

In a **Online Query Mechanism**, the *Online Certificate Status Protocol (OCSP)*[26] is one of the main standards. In the OCSP system, the user sends a status request to a OCSP *responder*, that processes the request and replies to the sender with *status information* about the queried certificate. An OCSP request contains the following fields:

- protocol version;
- service request;
- target certificate identifier;
- optional extensions;

The OCSP is a trusted entity. Therefore, to ensure the authenticity of the response, the OCSP responder digitally signs the response sent to the user. The response contains the *certificate identifier* and *response validity interval*, besides the certificate status value. In the certificate status field, the following indicators are possible:

- *good*: certificate has not been revoked, therefore is valid;
- *revoked*: certificate been revoked, temporarily or permanently;
- *unknown*: information about the status of the certificate could not be obtained;

Even thought OCSP tackles some of the shortcomings of a *Periodic Publication Mechanism*, like CRLs, there are still some limitations. The aforementioned *revocation delay* may still exist while using OCSP. From the scalability side it can generate an huge overhead in the OCSP Responder since clients ask for single certificates, even more in the case of high traffic services. Also, OCSP does not define how the necessary information is retrieved from the CRL repository by the OCSP responder.

The CA model has some inherent problems[27]:

- The CA needs to be a ”trusted” entity, but delegating the certification tasks to a single entity without having knowledge of the system internals could be considered ”blind trust”. Having an open implementation system and the possibility of public audits could lessen this problem;
- As the CA needs to identify the applicants before issuing the certificates, a proper mechanism to ensure the user true identity must be in-place.

2.3.2 Web Of Trust

In a *Web of Trust*[28] model, users trust a public key if it is obtained directly from the owner or a sufficient number of other trusted users recommend using the key. This recommendation is done by vouching for someone identity through the signing of their public keys.

Pretty Good Privacy (PGP) [29] is a standard that implements the web of trust model. The open-source implementation is *GNU Privacy Guard*[1].

Each PGP user keeps a *key ring*, where he stores his own public key and the public keys of other users. Each key ring entry contains the following fields:

- public key of user;
- User identifier of the public key owner;
- Validity signatures of the public key and the respective user IDs of the signers;
- Owner trust;
- Key legitimacy, also known as key validity.

---

[1] https://www.gnupg.org/

The *owners trust* field is set by the key ring owner and indicates the level of trust the key ring owner has in the public key owner to sign other users' public keys. It can take the values:

- *Ultimate*: assigned to key ring owners;
- *Complete*: the key ring owner trusts totally the public key owner to sign other public keys;
- *Marginal*: the key ring owner only trust marginally the public key owner to sing other public keys;
- *None*: the key ring owner don't trusts the public key owner to sign other public keys;
- *Unknown*: no information about the public key owner.

The *key legitimacy* field indicates the trust of the key ring owner of the authenticity of the public key. It is calculated by the number of signatures on the key and the owner-trust. It can take the values:

- *Complete*: the owner is certain that the public key belongs to the user identifier depicted in the entry;
- *Marginal*: the owner is marginally certain that the public key belongs to the user identifier depicted in the entry;
- *None*: the owner is not sure that the public key belongs to the user identifier depicted in the entry.

In order to be able to use the OpenPGP standard, public keys and respective signatures must be exchanged. This is possible by users exchanging directly their public keys or by using a *key server*. A key server is a public directory service that allows users to store and share public keys and their signatures.

2.4 Blockchain and the Bitcoin protocol

The blockchain is a distributed ledger and one of the key mechanisms behind the Bitcoin[30] cryptocurrency. It allows a set of nodes to achieve consensus about the state of a dataset, by leveraging a mechanism of proof of work.

At its core, a blockchain data structure consist of a linked list of blocks. Each of these blocks contain several transactions records that occurred between any two bitcoin entities. When a new transaction record is built, it is broadcast to all nodes in the network. Each node groups a list of transactions into a block, tries to find a proof-of-work for it and broadcasts the block to the network. Then, each node verifies the proof-of-work, and if successful, adds the block to the blockchain. This concept is called *mining*, and every node that can attach a new block to the blockchain receives an incentive in the form of newly minted bitcoins.

2.4.1 Block structure

A block is a container data structure that aggregates transactions for inclusion in the blockchain [31].

As seen in Figure 2.3 a block has several required fields:

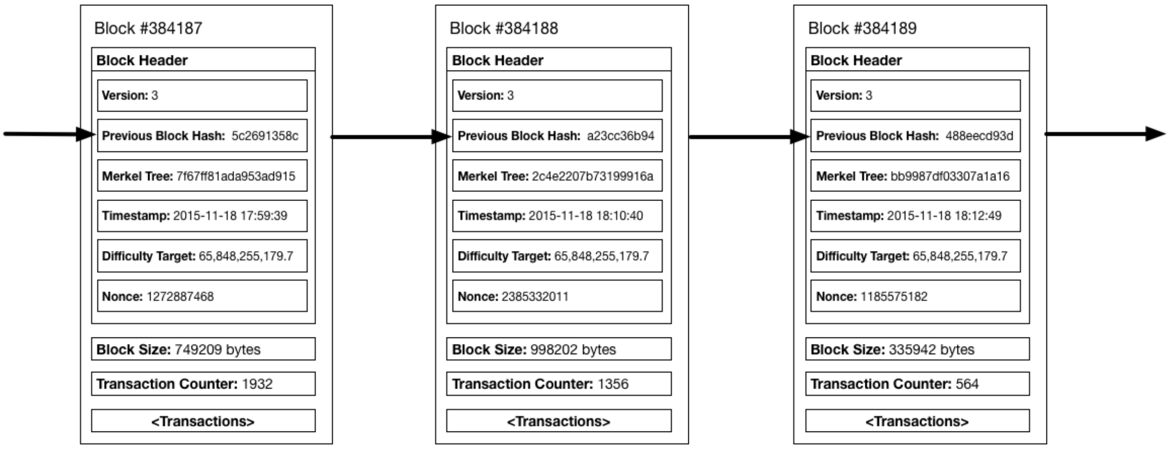

Block #384187
Block Header
Version: 3
Previous Block Hash: 5c2691358c
Merkel Tree: 7f67ff81ada953ad915
Timestamp: 2015-11-18 17:59:39
Difficulty Target: 65,848,255,179.7
Nonce: 1272887468
Block Size: 749209 bytes
Transaction Counter: 1932
<Transactions>
Block #384188
Block Header
Version: 3
Previous Block Hash: a23cc36b94
Merkel Tree: 2c4e2207b73199916a
Timestamp: 2015-11-18 18:10:40
Difficulty Target: 65,848,255,179.7
Nonce: 2385332011
Block Size: 998202 bytes
Transaction Counter: 1356
<Transactions>
Block #384189
Block Header
Version: 3
Previous Block Hash: 488eecd93d
Merkel Tree: bb9987df03307a1a16
Timestamp: 2015-11-18 18:12:49
Difficulty Target: 65,848,255,179.7
Nonce: 1185575182
Block Size: 335942 bytes
Transaction Counter: 564
<Transactions>

Figure 2.3: Blockchain structure and blocks content.

- **Block Size**: size of the block, in bytes, minus this field;
- **Block Header**: contains several block related metadata;
- **Transaction Counter**: number of transactions in the block;
- **Transactions**: transactions contained in the block.

One of the most important fields in the block is the *block header* field which contains the most important sets of metadata:

- **Version**: software version number;
- **Previous Black Hash**: previous block reference;
- **Merkle Root**: root of the merkle tree that contains the block's transactions;
- **Timestamp**: creation time of this block (using Unix Epoch);
- **Difficulty Target**: difficulty of the proof-of-work algorithm for this block;
- **Nonce**: counter used by the proof-of-work algorithm.

2.4.2 Merkle Trees

Each block stores a summary of all the transactions in the block in a multi-level data structure called *Merkle Tree*. In Figure 2.4 an merkle tree example is shown.

Basically, *Merkle Trees* are binary trees containing cryptographic hashes composed by hashing recursively the children nodes, using a bottom-top approach. This allows summarizing the contents of large data sets and provides a secure and efficient form of verification of the data set integrity.

Checking if a data element is included in a tree with N hashed elements takes at most $2 * log_2(N)$ calculations.

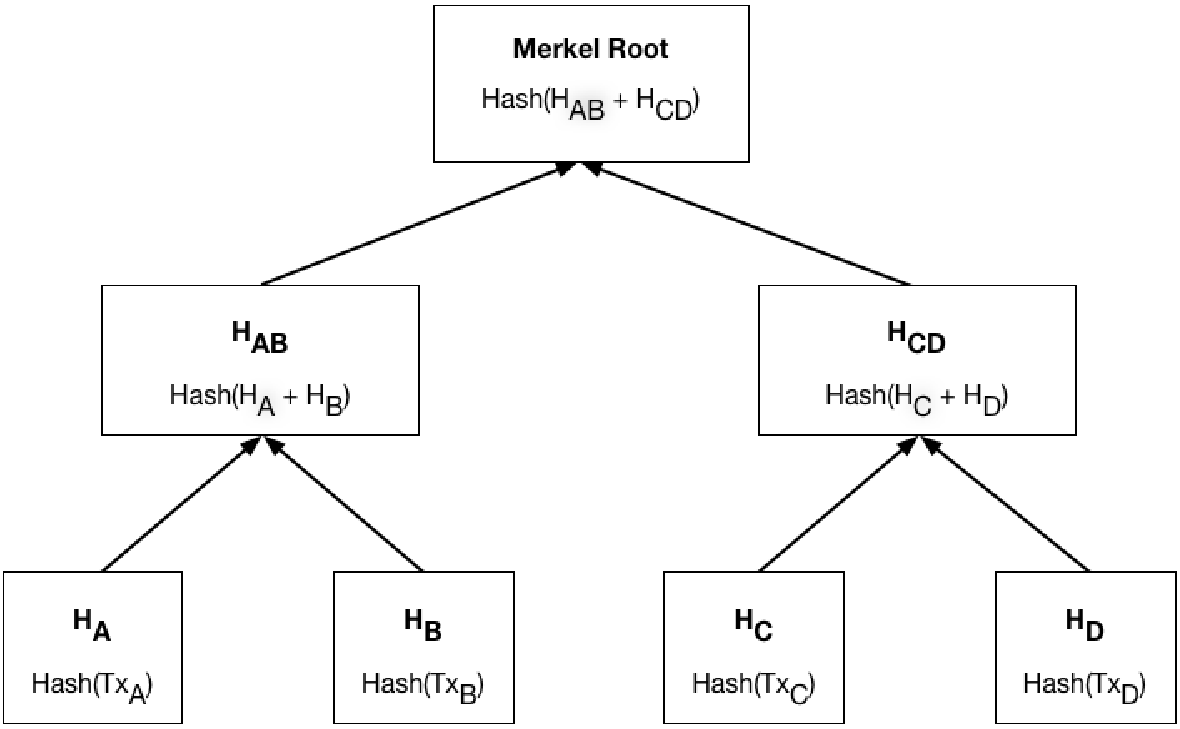


Figure 2.4: Merkle tree example

This data structure is an essential component in the Bitcoin protocol, since it prevents tampering of the transactions by malicious users. In order to successfully swap in fake transactions into the bottom of the *merkle tree*, it would be necessary to recalculate all the hashes of the nodes that are in the same sub-tree, up to the *merkle root*. Given that each block may contain several hundreds of transactions, this could potentially be a very expensive operation, and by the time it is complete, already new blocks that point to previous real block could have been mined.

2.4.3 Proof of work

In Bitcoin, the proof-of-work involves scanning for a value that when hashed together with block, the hash begins with a number of zero bits. This is calculated by incrementing a nonce in the block until a value is found that gives the block's hash the required number of zero bits.

Using this proof-of-work system, the problem of majority decision making is solved since this is a one-CPU-one-vote voting system.

2.4.4 Blockchain applications

Some systems were built on top of the blockchain by forking the Bitcoin project. *Namecoin*[2] is one of them, and is a decentralized name and information register. It can be used as a decentralized DNS or to save identity information like GPG keys, email information, etc.

---

[2] https://namecoin.info/

2.4.5 Smart Contracts

The concept of Smart Contracts[3][32] is the one of contracts: allowing to establish an agreement between several mutually suspicious parties. But smart contracts have the specificity of being possible to describe via program code and therefore, it is possible to enforce them programmatically and automatically. The Bitcoin protocol has a scripting system that is used for transactions and can be used to build a limited implementation of smart contracts.

2.4.6 Ethereum

Ethereum[4] is a protocol for building distributed applications on top of a blockchain, with a Turing-complete programming language, using the smart contract concept. It uses an internal cryptocurrency, *ether* which is similar to bitcoin, and is used to reward the computational resources used to process the smart contracts and securing the network.

In Ethereum there are two types of accounts:

- **externally owned accounts** are controlled by users in possession of a private key, and are able to send messages through transactions;
- **contract accounts** are controlled by *contract code*, which is executed every time it receives a message.

Each share a similar data structure, which contain the following fields:

- **Nounce**;
- **Current ether balance**;
- **Contract code** (if it is a contract account);
- **Storage**;

The communication model in Ethereum works through the concept of **messages** and **transactions**.

In Ethereum, a transaction is a signed data package that contains the message that a *externally owned account* wants to send. It contains the following fields:

- **Message recipient**;
- **Sender signature**;
- **Ether amount** to transfer;
- **Data field** (optional), that can be used to send user-defined data to the contract;
- **STARTGAS value**, value that represent the maximum computational steps the transaction can take;

---

[3] ”The Idea of Smart Contracts” - http://szabo.best.vwh.net/idea.html

[4] https://github.com/ethereum/wiki/wiki/White-Paper

```
contract DataStorage {
     uint dataStore;

     function set(uint a) {
        dataStore = a;
     }

     function get() constant returns (uint val) {
        return dataStore;
     }
 }
```

Figure 2.5: Simple data storage contract in Solidity language.

- **GASPRICE value**, the fee payed by the sender per computational step.

Ethereum prevents denial of service attacks that target resource consumption by using a unit to represent computational steps, called ”gas”. Each computational step that a user wants to execute requires the payment of a gas fee. This way, an attacker that tries to consume extra computational resources or create contract loops, will pay a gas fee proportional to the consumed resources.

Messages in Ethereum are virtual objects that can be exchanged between contract accounts, and only exist in the Ethereum Virtual Machine (EVM).

- **Message sender**;
- **Message recipient**;
- **Ether amount** to transfer;
- **Data field** (optional);
- **STARTGAS** value.

The propose of messages is to create a relationship between contracts, so it is possible for contracts to trigger the execution of other contracts. An important detail is that the assigned gas to the execution of the contract, must also take into account the gas consumed by execution of the other contracts required by the top contract.

Contract code

The contract code is written in a stack-based bytecode language called *EVM code*. The code execution is done in an infinite loop that repeatedly runs the operation in the current program counter position, until the end of the code, an error or return statement is reached.

The contracts code could be written in Solidity[5] a high-level contract language that can be compiled to EVM code. In Figure 2.5 an simple Solidity contract is shown. This contract defines two functions that allows anyone to save a value to the contract internal storage and retrieve it after.

---

[5] http://ethereum.github.io/solidity/

The Ethereum protocol, therefore allows to build several secure distributed applications by applying the abstract smart contract model: voting systems, decentralized file storage, Decentralized Autonomous Organization (DAO), identity and reputation systems, etc

Chapter 3

Architecture

This chapter describes the overall architecture of IDChain and outlines its main functional components. The main design goal is to allow to build a fully decentralized DHT-based system that is secure and can be closed to participation in a federated model, while minimizing trust in the intervenients. The main design goal is to allow to build a fully decentralized DHT-based system that is secure and can be closed to participation in a federated model, while minimizing trust in the intervenients. We introduce the main requirements in Section 3.1 and give a system architecture overview in Section 3.2. The remaining sections describe the in more detail the architecture of the solution and discuss the main design choices.

3.1 Requirements

3.2 Overview

The proposed architecture will consist of two independent, inter-connected systems:

- **Distributed Hash Table** based on Kademlia;
- **Decentralized Public Key Infrastructure** (DPKI) built on top of the blockchain.

In Figure 3.1 an high-level overview of the architecture is shown. In Section 3.3 and 3.4 an in-depth description of each individual component is given.

As can be seen in Figure **??** the two main components are the DHT and the DPKI.

The DPKI can be subdivided in three main components:

- **Blockchain**;
- **IDChain API**;
- **IDChain Application**;

In the scope of the current reTHINK project architecture, these components will be only used in a federated model, i.e the nodes in the DHT will all belong to the Service Providers (SP). Therefore it is

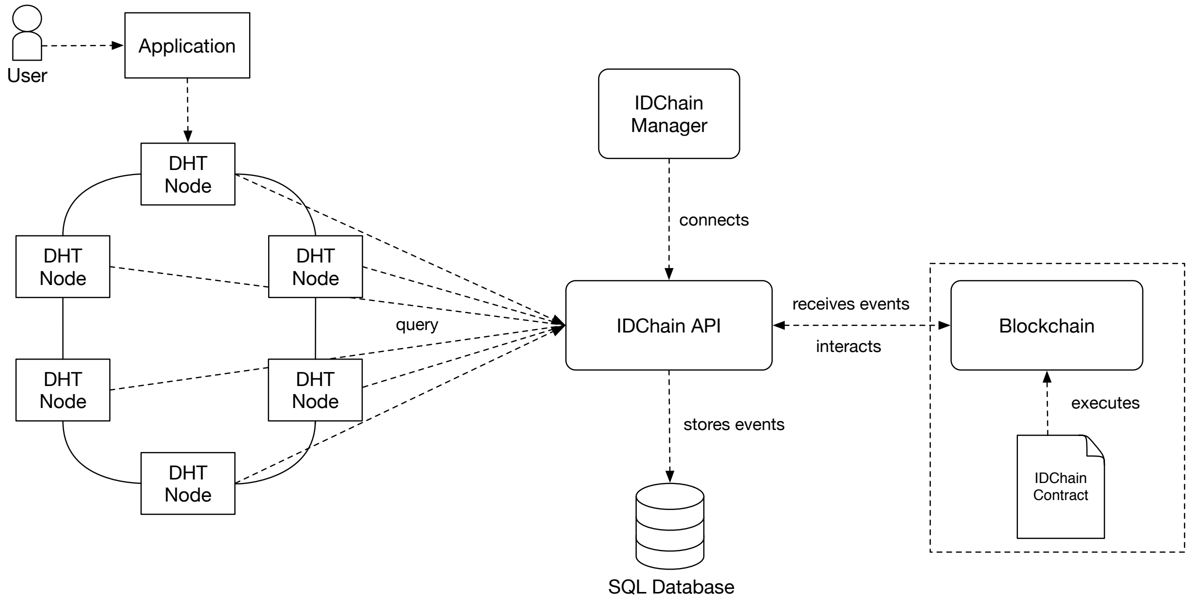

User
Application
DHT Node
DHT Node
DHT Node
DHT Node
DHT Node
DHT Node
query
IDChain Manager
connects
IDChain API
receives events
interacts
Blockchain
executes
IDChain Contract
stores events
SQL Database

Figure 3.1: Overview of solution architecture.

possible to assume some level of trust with the nodes, which opens the possibility to use a more traditional approach for managing the certificates, for exxample, a Certificate Authority. But, in a federated model we might want to minimize trust in other organizations or SP, in order to discourage fradulent activity or over-control of the system by one or several organizations. CAs have also the problem of adding an extra burden in organizations, since is not a totally automated system and still need some human intervention, mainly when accepting and signing Certificate Signing Requests (CSRs).

3.3 Distributed Hash Table

The DHT that will be deployed will be based on a Kademlia protocol implementation. This implementation should already ensure data republishing and replication.

Each node in the DHT will have a self-signed certificate that is needed to ensure a secure routing message exchange between nodes in the overlay network. When a node is routing a message through the overlay network, the peer connectivity is done using TLS connections with mutual authentication, so that the two peer certificates can be exchanged and verified. In order to verify the authenticity of the certificate, the peer identifier of the message sender should be equal to the peer identifier in the sender certificate. But this verification is not sufficient to guarantee the validity of the certificate, since a peer this way can impersonate several identities. So is necessary to implement a mechanism similar to certificate pinning. Therefore the peer should also check if there is a correspondence between the certificate fingerprint and the peer identifier registered in the blockchain.

3.4 Decentralized Public Key Infrastructure

Estabilishing Transport Layer Security (TLS) connections between nodes requires trusting the exchanged peers certificates during the TLS connection handshake. In a traditional setup the underlying PKI and CAs guarantee that the certificates are trustworthy, since the CAs sign the certificates. If the peer trusts the CA that signed off the certificate and have access to the CA certificate, it's able to verify the other peers certificates.

But if we want to minimize trust and build a fully decentralized model, trusting in a centralized entity like a CA defeats that purpose. Another mechanisms exist that try to decentralize this trust model, for instance, web of trust models, in systems like PGP, where users sign assertions for each other keys or certificates.

Still, most of the times PGP users still rely in a centralized mechanism, key servers, for distributing certificates or keys.

In order to obtain these certificates securely and verify their authenticity, in a totally decentralized manner without needing a CA, we are going to build a Decentralized Public Key Infrastructure (DPKI) using smart contracts in a blockchain.

The DPKI will have the following functionalities:

- Register and store certificates associating a node identifier with its certificate fingerprint;
- Revoke compromised certificates;
- Allow users to query the blockchain, in order to retrieve the associations between node identifiers and certificate fingerprint.

Since we want to build this DPKI mechanism on top of TLS, complementing it, we will use some mechanisms that are used in *Certificate Pinning*[1].

In the following sub-sections a more detailed overview of each piece of the DPKI architecture will be given.

3.4.1 IDChain Smart Contract

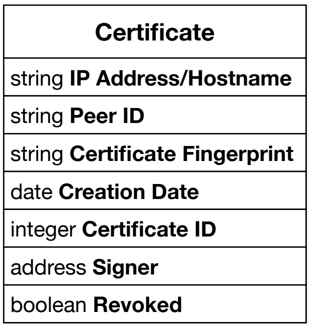
The logic and set of rules that compose the DPKI are stored in a blockchain smart contract. The smart contract that we gonna to create has the main objective of associate an peer identifier to a valid certificate, in a way that any system can easily query for and verify that association.

Trusted Certification

Creating this association is a sufficient condition to guarantee the validity and authenticity of a peer certificate, but we also need to be able to restrict and control the participation in the DHT, in order to be able to create a mechanism of defense against Sybil attacks. The starting point are the conclusions

---

[1] https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning



Certificate
string **IP Address/Hostname**
string **Peer ID**
string **Certificate Fingerprint**
date **Creation Date**
integer **Certificate ID**
address **Signer**
boolean **Revoked**

Figure 3.2: Smart contract *Certificate* structure fields.

drawn by Douceur[10] that trusted certification is the only approach that has the potential to eliminate Sybil attacks, however this certification relies on a centralized entity.

Smart Contracts and Autonomy

Although, with the advent of the blockchain technologies and smart contracts, is possible to build entities and organizations like the DAO [33], entities that are truly transparent, that can't be stopped or interrupted, and more importantly can't be corrupted or tampered. The rules and code that dictate the behavior of this entities, could be defined by a smart contract deployed in a blockchain.

This way, leveraging this technologies, is possible to build a system that decentralizes the trusted certification mechanism.

A good starting point is encoding a hard limit on the number of identifiers or certificates, that an entity can associate with their own blockchain address.

We can also opt by a soft limit or resource-based approach, for example, by imposing the payment of a value for each certificate association created, leveraging the cryptocurrency inherently associated with the blockchain system.

Smart Contract Logic

The main starting point the smart contract is the *Certificate* structure and fields. In Figure 3.2, is depicted the necessary fields in this structure. Note that is not necessary to save the full certificate metadata in the blockchain: the certificates are already exchanged in the TLS handshake between the nodes. Could be possible to save all the certificate metadata or even the full encoded certificate for searchability and navigability proposes, but blockchains are not appropriated to save large quantities of information. Not only blockchains have a maximum ammount of data that can be encoded in a block and transaction, also in the case of a blockchain as Ethereum, the cost of storing full certificates in the blockchain would be very high. Still, if this functionality was necessary and we wanted to maintain a decentralized system, a better approach would be store the full certificate in a **p2p!** (**p2p!**) filesystem like InterPlanetary File System (IPFS), and store in the *Certificate* structure the address hash of the certificate.

Since we are trying to build this DPKI under a federated model, is still necessary to define the trust mechanism in this system. So we are building the trust mechanism relying on a web of trust model built

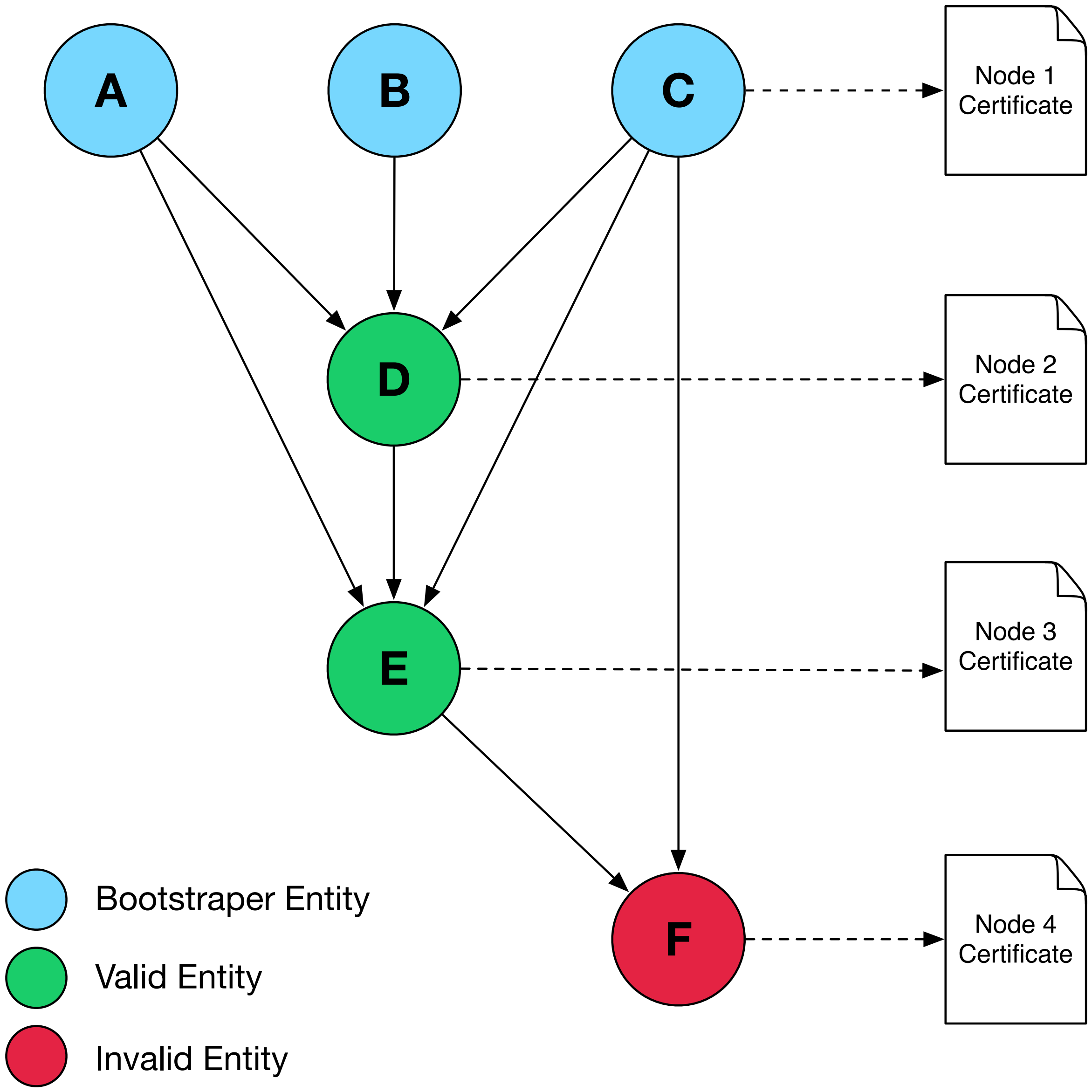

A
B
C
D
E
F
Node 1 Certificate
Node 2 Certificate
Node 3 Certificate
Node 4 Certificate
Bootstraper Entity
Valid Entity
Invalid Entity

Figure 3.3: Web of trust mechanism.

on top of the blockchain.

As is shown in Figure 3.3, a newly created entity to be accepted should be vouched by a minimum number of entities. One peculiar aspect of this architecture, consequence of the federated model, is that each entity in the system is able to create several node certificates. This can undermine smaller entities participating in the system, and could even allow one single entity to control the whole DHT, by generating unlimited node certificates. A combination of an hard limit, for example, a maximum number of certificates each entity can have, and a soft limit, i.e defining a cost, using the underlying cryptocurrency, for each certificate, could provider a equal and fair system for all the entities involved.

Defining the web of trust mechanism in the smart contract requires the creation of an *Entity* structure. In Figure 3.4 are defined the necessary fields. Is important to note that the entities addresses that an entity vouched for (*signed* field) and the entities addresses that vouched for our own entity (*signers*

Entity
string **Name**
address[ ] **Signers**
address[ ] **Signed**
Certificate[ ] **Certificates**
boolean **Valid**
boolean **Bootstraper**

Figure 3.4: Smart contract *Entity* structure fields.

field) are stored in this entity, because will be necessary to constantly calculate the state of each entity. If we considered a graph representation (usual in the web of trust model), this fields encode the inbound arrows and outbound arrows of each entity.

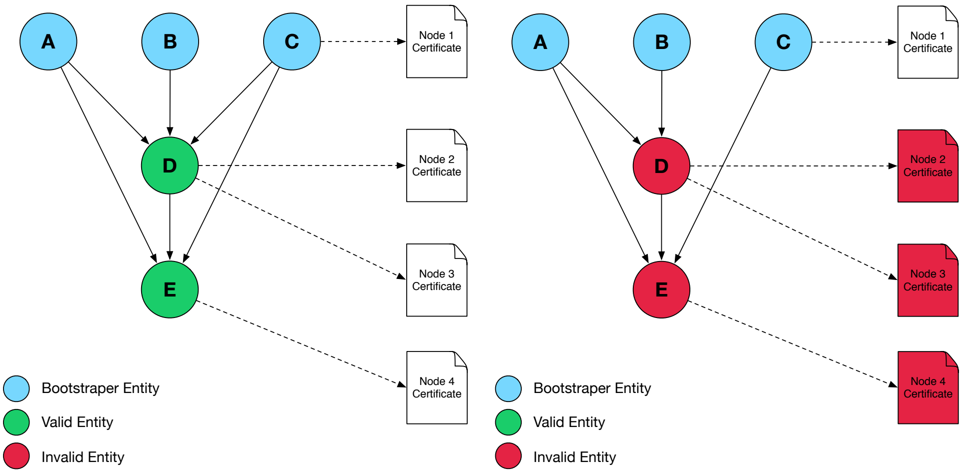
Another important aspect to take into account when building this smart contract is the bootstrapping process. In order to the web of trust mechanism to work is necessary to define the initial entities which are trusted. Since we are in an federated model, the entities that will deploy the smart contract could prearrange the entities that will be trusted from the beginning. Another possible solution, which we will use, is to give a trust status to the first *n* entities that will register in the smart contract. The entities that are given this initial trust will have a *true* value in the *bootstraper* field.

The minimum functionalities or functions that the smart contract should provider are:

- **Register a new entity** - this function should init a new entity associated with the blockchain account that called the function;
- **Create a new certificate** - generate a new certificate with the given fields in the blockchain, creating also the respective associations with the entity generating it;
- **Revoke the certificate** - allow the entity that generated the certificate to revoke it;
- **Vouch for entity** - should add signer address to the *signers* structure in the target entity, and add the target entity address to the *signed* structure in the signer entity;
- **Unvouch for entity** - should remove the source entity address from the *signers* structure in the target entity, and remove the target entity address from the *signed* structure in the signer entity;
- **Check entity validity** - after vouching or unvouching an entity, is necessary to check the target entity and its dependants. If any of the entities along the chain of trust doesn't have the minimum required vouches, it should be considered invalid;

Web of trust scenarios

In some aspects the web of trust mechanism that we are going to implement is different from the more well-known web of trust systems. One of these aspects, as shown in Figure 3.5(a), is that a single entity could create several node certificates. As said before, this is aligned with the fact that, as we are working


A
B
C
Node 1 Certificate
D
Node 2 Certificate
E
Node 3 Certificate
Node 4 Certificate
Bootstraper Entity
Valid Entity
Invalid Entity
A
B
C
Node 1 Certificate
D
Node 2 Certificate
E
Node 3 Certificate
Node 4 Certificate
Bootstraper Entity
Valid Entity
Invalid Entity

(a) Scenario A (b) Scenario B

Figure 3.5: In Scenario A bootstraper entities *A, B and C* vouch for entity *D*, which vouches for entity *E* in conjunction with entities *A* and *C*. In Scenario B, entity *C* unvouches entity *D* which is considered invalid and therefore also entity *E* is considered invalid.

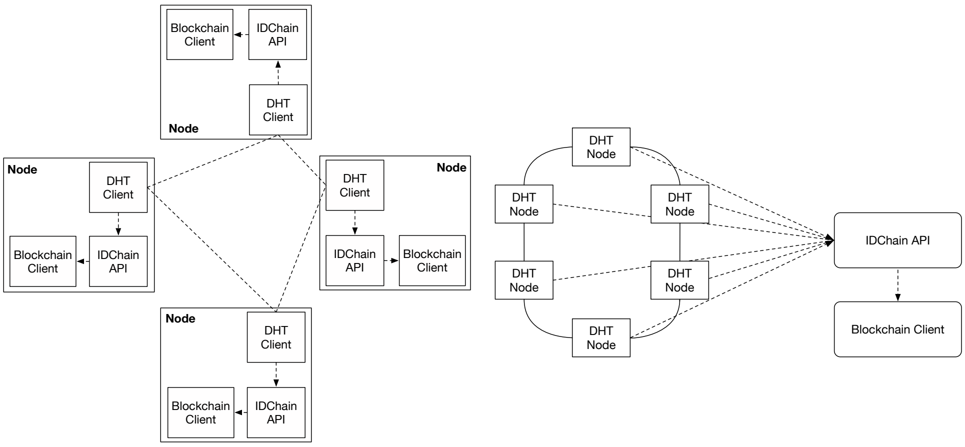
in a federated model, we may want that each SP can have multiple nodes, in order to the DHT to be scalable. The valid state of each certificate is determined by its entity state, if a entity has the minimum number of vouches required by the system, then the certificate created by the entity are considered valid. This is depicted in Figure 3.5(a), where *Entity D* has two certificates: *Node 2 Certificate* and *Node 3 Certificate*. These certificates are considered valid because the entity that created them, *Entity D*, has the minimum number of vouches required, and therefore is considered a valid entity.

If one entity loses one vouch and its number of vouches drops below the minimum threshold, then is necessary to verify every descendant of the entity, and invalidate any entity that this way does not have the minimum number of vouches. This could be verified in Figure 3.5(b), *Entity C* unvouches *Entity D* which is rendered invalid (considering a minimum of three vouches), and when verifying its descendants *Entity E* also doesn't have the minimum number of vouches required, and therefore is rendered invalid also. Is worth mentioning that the certificates of *Entity D* and *Entity E*, as mentioned before, are also considered invalid.

3.4.2 IDChain API

In order to provide a better and universal interface to interact with the IDChain smart contract and the blockchain, we will build a RESTful API to facilitate access to the functionalities. This API will run on the SP network, and can be deployed in two ways: a single API instance for all the nodes controled by the SP as shown in Figure 3.6(a), or a multiple API instances, one for each node, running in the same server, shown in Figure 3.6(b).

The API will have the classic components usually associated with a RESTful service: every endpoint returns JSON documents and is possible to address every stored resource (certificates, entities, transactions, etc) in the system. The endpoints that will be available are presented in Table 3.1 An advantage


Blockchain Client
IDChain API
DHT Client
Node
Node
DHT Client
Blockchain Client
IDChain API
DHT Client
Node
IDChain API
Blockchain Client
Node
DHT Client
Blockchain Client
IDChain API
DHT Node
DHT Node
DHT Node
DHT Node
DHT Node
DHT Node
IDChain API
Blockchain Client

(a) API Multi-deploy

(b) API Single-deploy

Figure 3.6:

of building this API to access the IDChain functionalities, is that allows to build applications and services that easily interact with it through **HTTP!** (**HTTP!**) while being language-agnostic, easing the access to the system. An example of such an application that uses the IDChain API is the IDChain Management App, that we will describe in greater detail in Subsection 3.4.3.

The IDChain API will be backed up by a relational database (using **SQL!** (**SQL!**)) where all the transactions related with the system will be stored. The reasoning behind this mainly due to performance: storing all this information in a local **SQL!** database will remove the necessity of constantly querying the blockchain. Also using a relational database allows to structure all the transactions and data associated with the system, which enables more powerful queries in a simpler way. For example, obtaining all the certificates associated with an entity, or get the blockchain transaction where a specific revocation was made. This kind of associations and structure is harder to obtain when using the blockchain client directly, since the client only deals with transactions at a lower level, without attaching a meaning at a smart contract basis to each transaction. In order to store the data in the database, we will use the blockchain *events*. Is possible to trigger events in smart contracts that, when are executed in a transaction context, will notify the blockchain client, i.e. when a client attaches a valid block to the blockchain the client will be notified of this event. This allow to mirror the transaction information related with our smart contract to the relational database.

3.4.3 IDChain Management Application

We will also build an application that allow to do all the operations related with the IDChain system. Will be an web application, that will interact directly with the IDChain API. The main functionalities will revolve around entity and certificate management: view entities vouches, create new node certificates, view the transactions related with each entity, etc. The main inspiration for this system are the web-based *blockchain explorers*[2] that exist for blockchains like Ethereum and Bitcoin. The web application will be built using an Single Page Application (SPA) architecture, leveraging browser Javascript frameworks.

---

[2] https://blockexplorer.com/

Endpoint	**Functionality**
GET /certificates	List all certificates.
GET /certificate/{id}	Get the certificate with the specified *id*.
POST /certificate	Create a new certificate-peer association.
GET /entities	List all the registered entities.
GET /entity/{id}	Fetch the entity with the specified *id*.
GET /entity/{id}/certificates	Fetch all the certificates created by the entity with the specified *id*.
GET /entity/{id}/transactions	Fetch all the transactions associated with the entity with the specified *id*.
GET /entity/{id}/signatures	List the entities that the entity with the specified *id* vouched.
GET /entity/{id}/signed_by	List the entities that vouched for the entity with the specified *id*.
GET /peer/{id}	Fetch the certificate entry associated with the specified peer *id*.
GET /signatures/{target}	Vouch for the *target* entity.
DELETE /signatures/{target}	Unvouch the *target* entity.
GET /transaction/{id}	Fetch the transaction with the *id*.
GET /accounts	Fetch the accounts configured in the current blockchain client.

Table 3.1: IDChain API specification

This will enabled us to recreate a desktop application fell, keeping the web application convenience and ease of access.

3.5 Overlay network processes

3.5.1 Node bootstrap and registration

In order to participate in the DHT, first the SP which wants to deploy the node must create a blockchain account/wallet, and then initialize his entity (one for each account) in the IDChain system. The entity then should be validated by the other entities vouches. Then should create the node's X509 self-signed certificate, and register the certificate fingerprint in the blockchain through the IDChain smart contract.

Finally the node will enter the DHT network using a node identifier equal to the one described in the registered certificate.

3.5.2 Node message routing

Every time a node needs to send a message to another node, it will perform the TLS connection handshake. The node will receive the destination node certificate, and will send his own certificate to the

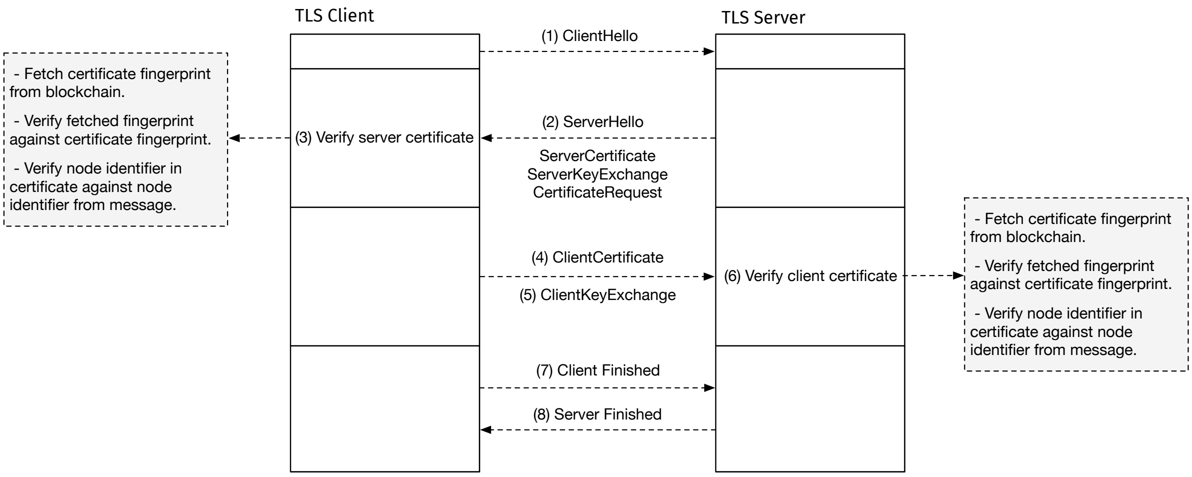

TLS Client
TLS Server
(1) ClientHello
- Fetch certificate fingerprint from blockchain.
- Verify fetched fingerprint against certificate fingerprint.
- Verify node identifier in certificate against node identifier from message.
(3) Verify server certificate
(2) ServerHello
ServerCertificate
ServerKeyExchange
CertificateRequest
(4) ClientCertificate
(5) ClientKeyExchange
(6) Verify client certificate
- Fetch certificate fingerprint from blockchain.
- Verify fetched fingerprint against certificate fingerprint.
- Verify node identifier in certificate against node identifier from message.
(7) Client Finished
(8) Server Finished

Figure 3.7: Additional steps added in TLS handshake verification.

destination node, since we will use TLS mutual authentication. Each node will fetch the certificate information from the blockchain through the IDChain API by the destination node identifier, by issuing a *GET* request to the */peer/*{*id*} endpoint.

Will be necessary to add an extra-step to the TLS handshake, to verify the validity of the exchanged certificates. As depicted in Figure 3.7 is necessary to verify the following:

- The fingerprint of the exchanged certificate is equal to the fingerprint registered in the blockchain;
- The node identifier registered in the certificate is equal to the node identifier.

3.5.3 Eclipse and Sybil attacks defense

The mitigation of possible Sybil and Eclipse attacks leverage the blockchain cryptocurrency.

The execution of the certificate creation process in the blockchain will require the payment of a value in ether, defined by the contract creator, which will be saved in the *contract account* balance. If an attacker tries to launch a Sybil attack, he will need to pay an enormous amount of ether, making the attack infeasible. This solution will also help mitigate possible Eclipse attacks, since preventing Sybil attacks is a constraint to launch Eclipse attacks. Coupled with the Kademlia k-bucket replacement policy, where new nodes are only added if a bucket is not full and parallel routing, an efficient defense against Eclipse attacks is achieved.

3.5.4 Data replication

When a DHT node receives a request to store user data, is necessary to ensure data replication, in order to achieve data availability. The implemented DHT will use TomP2P direct replication method. In this method a node constantly republishes its own content to other nodes, and its responsible for its content. In case of node failure, the content may timeout and be removed, but since the DHT node run in a SP, a fast restart of the DHT node is assumed.

3.6 Comparison with a CA

Using a blockchain to build a PKI, is possible to minimize the trust, relatively to a centralized solution, like a CA. Due to the nature of the blockchain as an immutable transactional database, is possible to guarantee the validity of each transaction, through inspection of the past ones. This immutability property also allows to perform audits to the blockchain, in order to verify the integrity of the data stored. The biggest disadvantage of the blockchain is the computational resources needed to guarantee its proper operation and security.

Chapter 4

Implementation

4.1 Implementation Options

Neste secção devem apresentar as opções de implementação que tinham ao vosso dispor, avaliá-las e justificar a escolha que realizaram. Isto pode englobar:

- Simuladores
- Linguagens e ambientes de programação
- Sistemas operativos
- Hardware

Fica sempre bem na avaliação colocar uma tabela com as características pretendidas e as que são satisfeitas pelas várias opções (tipo catálogo com as características dos automóveis). A escolha deve surgir naturalmente, com base na opção que tem mais cruzes…

4.2 Architecture

Nesta secção devem explicar como implementaram a vossa solução, apresentando as simplificações que efectuaram, face ao modelo inicialmente previsto. As simplificações devem ser devidamente justificadas. Se for possível, devem indicar que estas não põem em causa as contribuições da tese. Podem ainda descrever os principais problemas que tiveram e a forma como os abordaram e resolveram.

Se estiverem a usar um simulador devem:

- explicar o funcionamento do simulador
- explicar as alterações e modelos que desenvolveram no simulador e que permitem validar a vossa ideia

Se estiveram a desenvolver SW, sem simulador devem:

- explicar os módulos, interfaces, estruturas de dados, etc...

Sempre que possível, ilustrem a arquitectura com figuras que demonstrem a evolução face à arquitectura da secção anterior. Isto é, usem as figuras anteriores e façam as modificações necessárias à obtenção da arquitectura do protótipo. ...

Chapter 5

Evaluation

5.1 Tests Objectives

Nesta secção devem descrever os objectivos dos testes que realizaram, explicitando a razão pela qual os testes são relevantes face à validação das contribuições da tese.

5.2 Tests Scenarios

Nesta secção devem descrever o cenário de teste, incluindo, por exemplo, a definição da rede, o modelo de tráfego, as características de cada elemento.... A descrição deve ser feita, de forma a que os testes possam ser reprodutíveis. Se os testes forem feitos em ambiente real devem ser descritas as características dos equipamentos, memória, CPU, disco, SO, etc....

Devem também descrever as características das experiências, do ponto de vista estatístico. Número de testes realizados, grandezas que vão ser medidas, formas de medição dos valores, etc...

Sempre que possível, ilustrem o cenário de testes com figuras e com tabelas, que descrevam sucintamente o modelo.

5.3 Test Results

Nesta secção devem apresentar os resultados dos testes, quer sobre a forma de tabelas, quer sobre a forma de gráficos. As tabelas e os gráficos devem ser apresentados e depois analisados, detalhadamente. . . .

Here is an example of a table 5.1.

Table 5.1: Table caption

item 1	item 2
item 3	item 4

Chapter 6

Conclusions

6.1 Summary

Neste secção deve-se fazer o resumo do trabalho efectuado, retomando a ideia, as contribuições definidas e a forma como estas se materializaram

6.2 Achievements

The major achievements of the present work . . . Nesta secção devem ser retiradas conclusões do trabalho realizado, em face dos resultados obtidos.

6.3 Future Work

Nesta secção devem identificar o trabalho futuro, sob duas perspectivas:

- o trabalho que resulta directamente dos problemas que a vossa proposta criou, ou não conseguiu resolver
- o trabalho que resulta da evolução do sistemas

Appendix A

Vector calculus

Use to include images/diagrams tables which are important but were to big to include in the main tex.

Note that in no case the document can exceed a total of 100 pages.

Some equations examples:

A.1 Vector identities

$$\nabla \times (\nabla \phi) = 0 \tag{A.1}$$

$$\nabla \cdot (\nabla \times \mathbf{u}) = 0 \tag{A.2}$$

Bibliography

[1] Paulo Chainho, Kay Haensge, S.D., Maruschke, M.: Signalling-on-the-fly: Sigofly. In: 18th International Conference on Intelligence in Next Generation Networks, ICIN 2015, Paris, France, February 17-19, 2015. (2015) 1–8

[2] Gupta, A., Awasthi, L.K.: Peer-to-peer networks and computation: Current trends and future perspectives. Computing and Informatics **30**(3) (2011) 559–594

[3] Lua, E.K., Crowcroft, J., Pias, M., Sharma, R., Lim, S.: A survey and comparison of peer-to-peer overlay network schemes. IEEE Communications Surveys and Tutorials **7**(2) (2005) 72–93

[4] Gnutella, R.F.C.: The Gnutella Protocol Specification v0.4 (2004)

[5] Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM '01) (2001) 149–160

[6] Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Middleware 2001 **2218**(November 2001) (2001) 329–350

[7] Maymounkov, P., Mazières, D.: Kademlia: A peer-to-peer information system based on the xor metric. In: Revised Papers from the First International Workshop on Peer-to-Peer Systems. IPTPS '01, London, UK, UK, Springer-Verlag (2002) 53–65

[8] Urdaneta, G., Pierre, G., Steen, M.V.: A survey of DHT security techniques. ACM Computing Surveys **43**(2) (2011) 1–49

[9] Castro, M., Druschel, P., Ganesh, A., Rowstron, A., Wallach, D.S.: Secure routing for structured peer-to-peer overlay networks. Proceedings of the 5th symposium on Operating systems design and implementation OSDI 02 **36**(December) (2002)

[10] Douceur, J.: The Sybil attack. Peer-to-peer Systems (2002) 251–260

[11] Wang, L., Kangasharju, J.: Real-world sybil attacks in BitTorrent mainline DHT. GLOBECOM - IEEE Global Telecommunications Conference (2012) 826–832

[12] Wang, H.W.H., Zhu, Y.Z.Y., Hu, Y.H.Y.: An efficient and secure peer-to-peer overlay network. The IEEE Conference on Local Computer Networks 30th Anniversary (LCN'05)l (2005)

[13] Bazzi, R.a., Konjevod, G.: On the establishment of distinct identities in overlay networks. Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing - PODC ’05 (2005) 312

[14] Baumgart, I., Mies, S.: S/Kademlia: A practicable approach towards secure key-based routing. Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS **2** (2007)

[15] Yu, H., Kaminsky, M., Gibbons, P.B., Flaxman, A.D.: SybilGuard: Defending against sybil attacks via social networks. IEEE/ACM Transactions on Networking **16**(3) (2008) 576–589

[16] Yu, H., Gibbons, P.B., Kaminsky, M., Xiao, F.: SybilLimit: A near-optimal social network defense against sybil attacks. IEEE/ACM Transactions on Networking **18**(Figure 1) (2010) 885–898

[17] Margolin, N.B., Levine, B.N.: Informant: Detecting Sybils Using Incentives. Financial Cryptography and Data Security **4886/2008** (2008) 192–207

[18] Singh, A., Ngan, T.W., Druschel, P., Wallach, D.S.: Eclipse attacks on overlay networks: Threats and defenses. Proceedings - IEEE INFOCOM **00**(c) (2006)

[19] Rivest, R.L., Shamir, a., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM **21**(2) (1978) 120–126

[20] Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory **22**(6) (1976) 644–654

[21] Choudhury, S.: Public key infrastructure : implementation and design. M & T Books, New York, NY (2002)

[22] Entrust: Trusted public-key infrastructures. Technical report, Entrust - Secure Digital Entities & Information (aug 2000)

[23] Buchmann, J.A., Karatsiolis, E., Wiesmaier, A.: Introduction to Public Key Infrastructures. Volume 53. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)

[24] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard) (May 2008) Updated by RFC 6818.

[25] ITU, I.T.U.: Information technology - open systems interconnection - the directory: Public-key and attribute certificate frameworks. Series X: Data Networks, Open System Communications and Security Directory E 38895, International Telecommunication Union (oct 2012) ITU-T Recommendation X.509.

[26] Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., Adams, C.: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960 (Proposed Standard) (June 2013)

[27] Ellison, C., Schneier, B.: Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure. Computer Security Journal **16**(1) (2000)

[28] Caronni, G.: Walking the Web of trust. Proceedings IEEE 9th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2000) (2000) 153–158

[29] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., Thayer, R.: OpenPGP Message Format. RFC 4880 (Proposed Standard) (November 2007) Updated by RFC 551.

[30] Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System. Consulted (2008) 1–9

[31] Antonopoulos, A.M.: Mastering Bitcoin: Unlocking Digital Crypto-Currencies. 1st edn. O'Reilly Media, Inc. (2014)

[32] Miller, M.S., Cutsem, T.V., Tulloh, B.: Distributed electronic rights in javascript. In: ESOP'13 22nd European Symposium on Programming. (2013)

[33] Merkle, R.C.: Daos, democracy and governance. Cryonics Magazine **37** (jul 2016) 28–40