# Secure DHT with Blockchain technology

**Nuno Filipe Mateus Nogueira**

Thesis to obtain the Master of Science Degree

**Telecommunications and Informatics Engineering**

**Examination Committee**

Chairperson: Prof. Dr. Paulo Jorge Pires Ferreira
Supervisor: Prof. Dr. Ricardo Jorge Feliciano Lopes Pereira
Members of the Committee: Prof. Dr. Miguel Pupo Correia

**October 2016**

# Acknowledgments

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

# Abstract

With an increased usage of Distributed Hash Tables (DHT) as a basis for building scalable Peer-to-Peer (P2P) systems the security considerations and close participation in DHTs systems are still major concerns.

One system that was built using a DHT system is the Global Registry component of the European funded research project reTHINK. With the necessity of securing the DHT system of this component, and at the same time reducing trust between participants in the DHT, we present IDChain. The IDChain system is a Decentralized Public Key Infrastructure (DPKI) built on top of the Ethereum blockchain, which allows Service Providers (SP) to associate nodes with an identity, therefore providing access control and secure communications between nodes, in a decentralized fashion. Our approach comprises the creation of a smart contract in the Ethereum blockchain which mimics a Web of Trust model, allowing entities (SP) to register their unique nodes' identifiers and certificates, hence enabling connection establishment between nodes through Transport Layer Security (TLS). In order to ease the integration and management of the system, we also built a RESTful API and a web application.

This document surveys the current state of the art of P2P systems and DHTs security mechanisms. Our proposal, which consists of the IDChain system and a Certificate Authority (CA) based system – for comparison proposes – is presented in detail and validated through performance, security and monetary cost evaluation. We show that the IDChain proof-of-concept is performant and secure, therefore presenting a valid alternative to a CA-based solution.

**Keywords:** Peer-to-Peer, Distributed Hash Table, Blockchain, Public-Key Infrastructure, Web-of-Trust, reTHINK

# Resumo

Com um aumento no uso de Distributed Hash Tables (DHT) como uma base para construir sistemas Peer-to-Peer (P2P) escaláveis, as considerações de segurança e o fecho da participação nestes sistemas são ainda preocupações.

Um sistema que foi construído tendo como base uma DHT foi o componente Global Registry, pertencente ao projecto de investigação europeu reTHINK. Tendo em conta a necessidade de melhorar a segurança da DHT deste componente, reduzindo o nível de confiança entre os participantes, apresentamos o sistema IDChain. O sistema IDChain é uma Infraestrutura de Chaves Públicas Descentralizada, construída sobre o Ethereum, que permite aos Provedores de Serviço (PS) associarem nós da DHT a identificadores e certificados, providenciando controlo de acesso e estabelecimento de comunicações seguras entre os nós, de uma forma descentralizada, utilizando o protocolo Transport Layer Security (TLS). A nossa abordagem recorre à criação de um contracto inteligente no Ethereum, que utiliza um modelo Web of Trust, e que pode ser acedido através de uma API e aplicação web.

Este documento avalia o actual estado da arte de sistemas P2P e os mecanismos de segurança das DHTs. A nossa proposta, que consiste no sistema IDChain e num outro sistema baseado em Autoridades de Certificação (AC) – para efeitos de comparação – é apresentada em detalhe e validado através de uma avaliação de desempenho, segurança e custo monetário. Demonstramos que a prova de conceito do sistema IDChain tem um bom desempenho e é seguro, sendo então uma alternativa válida a uma solução baseada em ACs.

**Palavras-chave:** Peer-to-Peer, Distributed Hash Table, Blockchain, Infraestrutura de Chaves Públicas, Web of Trust, reTHINK

# Contents

# List of Figures

# List of Tables

# List of Acronyms

**API** Application Programming Interface

**CA** Certificate Authority

**CRL** Certificate Revocation List

**CSR** Certificate Signing Request

**DAO** Decentralized Autonomous Organization

**DER** Distinguished Encoding Rules

**DHT** Distributed Hash Table

**DLT** Distributed Ledger Technology

**DPKI** Decentralized Public Key Infrastructure

**EVM** Ethereum Virtual Machine

**GUID** Global User Identifier

**HTTPS** Hyper Text Transfer Protocol Secure

**HTTP** Hyper Text Transfer Protocol

**Hyperties** Hyperlinked Entities

**IPFS** InterPlanetary File System

**IoT** Internet of Things

**M2M** Machine-to-Machine

**MAC** Media Access Control

**MITM** Man-in-the-middle

**MSP** Membership Service Provider

**NIO** Non Blocking IO

**NPM** Node Package Manager

**OCSP** Online Certificate Status Protocol

**ORM** Object-Relational Mapping

**OTT** Over-the-top

**P2P** Peer-to-Peer

**PEM** Privacy Enhanced Mail

**PGP** Pretty Good Privacy

**PKI** Public Key Infrastructure

**POW** Proof of Work

**RTT** Round-Trip Time

**SPA** Single Page Application

**SP** Service Provider

**SQL** Structured Query Language

**SSH** Secure Shell

**SSL** Secure Sockets Layer

**UI** User Interface

**VM** Virtual Machine

**TLS** Transport Layer Security

# Chapter 1

# Introduction

## 1.1 Motivation

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 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 blocks new entrants.

On the other hand, we have the worldwide Telco ecosystem that provides a highly reliable service and strong trustful identity. Since it 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, which means that 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. The main goal is to achieve a framework that enables developers to create communication-based applications, allowing users from different reThink-based applications.

This dynamic trusted relationship is created by using Hyperlinked Entities (Hyperties), a web microservice paradigm, that enables 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

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

common API enabling communication between different hyperties from different service providers. The hyperty concept permits 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. The hyperties are maintained by Service Providers (SPs) and are loaded to the users device.

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:

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

The Registry service is sub-divided in three components: *Global Registry*, *Domain Registry* and *Local Registry*. The *Global Registry* is a key-value store based on DHT technology that stores user identifiers in hyperty services, indexed by a Global User Identifier (GUID). It lists domain-dependent identities owned by a user in the reTHINK system. By resolving a GUID, it is possible to obtain these domain-dependent identities of each user. The *Domain Registry* is run by SPs and allows to lookup users hyperty instances information using the domain-dependent identities mentioned above. It uses a client-server model, which allows to handle a high data update rate. The *Local Registry* component runs in the device runtime and manages hyperty instances running in the runtime and contacts the Global and Domain Registries.

## 1.2 Problem Statement

The *Global Registry* is based on a DHT, a Peer-to-Peer (P2P) distributed system that usually is used in public-facing services and therefore is open to participation, allowing any node to join the network. One example is the Mainline DHT, that powers the BitTorrent network.

Several private and enterprise uses of DHT system exist, for instance, the Dynamo[2] key-value system from Amazon, or the Cassandra[3] database. These systems use DHTs as a building-block of their architecture for tasks that require multi-node coordination, as data sharding, for example, therefore not exposing the DHT directly. These solutions are also closed-source and don't have their source code available publicly.

The reTHINK framework is also inherently a federated solution, consequently it isn't open to participation from outside the SPs organization, but it should allow the entry of new nodes from any of SPs part of the federation. This is an important aspect, because as opposed to a private solution where we have

full control of all the infrastructure, in a federated solution we don't have full control and it is necessary to have additional infrastructure to create trust relations and verify identities between the SPs DHT nodes.

## 1.3 Proposed Solution

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

In order to accomplish this goal, a set of objectives need to be achieved:

- **Mitigate common DHT attacks**- several different types of attacks targetting the DHT exist. An in-depth analysis of each one of them and existing solutions is necessary;
- **Guarantee communication privacy, integrity and authenticity**- it is crucial to secure the DHT nodes communication;
- **Close DHT participation**- provide mechanisms to allow access-control nodes in the DHT;
- **Deal with nodes compromise**- allow to easily revoke access to compromised nodes or identities;
- **Minimize trust between participants**- even though our system can not tolerate maleficent SP nodes, we want to minimize the trust between SP nodes so that it isn't possible for a couple of SPs nodes to monopolize the DHT system;
- **Maintain system decentralized**- solve all the aforementioned problems, maintaining a decentralized architecture, which should also minimize trust between SPs;
- **Decouple the DHT system from the Global Registy**- despite the main use case of the DHT is the Global Registry, we want to decouple the DHT by providing a simple API that permits any application to be built on top of it.

We will build one DHT with two interchangeable mechanisms that lets us achieve the aforementioned objectives.

The first solution is based on a more classic approach to establish secure communications by using a Certificate Authority to issue certificates assigning identities to each used DHT identifier, granting us the mechanisms to perform access-control to DHT nodes and provide secure communications between nodes.

The second solution we will present is a novel approach that uses blockchain technology, which we named IDChain. Using the blockchain we will build a Decentralized Public Key Infrastructure (DPKI) that will permit us to issue unique identities and certificates to nodes across all SPs. This solution provides the same guarantees of the previous presented solution, while also allowing us to maintain a decentralized system and minimize the trust between participants.

These two solutions will have as basis a DHT that doesn't provide secure communications between nodes nor access-control. This is presented as a complementary mechanism which we will use to benchmark our two security mechanisms.

## 1.4 Outline

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

- **Chapter 1** presents the motivation, background and proposed solution.
- **Chapter 2** describes the state of the art of the technologies used in the solution architecture.
- **Chapter 3** describes the system requirements and the architecture of the solution.
- **Chapter 4** describes the implementation of the solution and the technologies chosen.
- **Chapter 5** describes the evaluation tests performed and the corresponding results.
- **Chapter 6** summarizes the work developed and future work.

# Chapter 2

# Related Work

This section addresses 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

P2P networking 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). [4] P2P systems networks are usually categorized as *Structured* or *Unstructured* [5].

### 2.1.1 Unstructured

In an Unstructured overlay network, nodes join the network by connecting to other nodes without prior knowledge of the topology, therefore creating a random network structure. Peers search for content using flooding as a querying mechanism to other nodes' content. Despite being resilient for locating highly replicated content and to a high churn rate, the flooding mechanism is not suitable for locating 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[6] 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 are also assigned unique identifiers from the same identifier space, called *keys*, by applying a cryptographic

hash function to the data. This allows for the creation of an index, where each key identifies the position of the corresponding dataset in the network, therefore making it 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, which are necessary to forward routing messages across the overlay network until they reach the destination node. Several DHT-based solutions exist which implement different overlay network structure and routing schemes.

Chord [7] and Pastry [8] 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 [9] maps nodes into a balanced binary tree recurring to the 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 has two useful properties:

- It is **unidirectional**, meaning 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 it is possible to do *caching* of keys;
- It 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, first reaching 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 they are not present. These 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 chosen such that it is very unlikely that all nodes fail within an hour.

Every message a node receives, triggers an update in the k-bucket for the sender identifier. If there is available space in the k-bucket, the new identifier is inserted at its tail – or if already present – moved to the tail. If the identifier is new and the bucket is full, the oldest element is pinged. If it responds, the node is 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 provides 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* joins 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 its 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 needs data republishing.

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

## 2.2 DHT security considerations

There are several security considerations [10] 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

Request routing 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 its own routing tables and update them accordingly, there are multiple attack vectors that an attacker can take advantage of [11]:

*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 [12]. 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[13].

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 [12].

One proposed solution is to use certified node identifiers[11] issued by a trusted 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 certificates must be bought from the CA entity.

Even though 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.

#### Network characteristics

One proposed solution that uses network characteristics is the *net-print*[14] 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.[15] 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 in which a group of identities reside. By assuring that, for instance, in two separate groups of identities, if any two identities chosen from different groups are distinct, then it is possible 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 an upper-bound on the distance *d(A,B)* between the points. This solution takes into account the following assumptions:

- the distance between two points satisfies 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 instance, 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, guaranteeing that identities in different groups are not controlled by the same entity. The major problem with this system is that it can be easily circumvented, if an attacker uses 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[16] 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 represents 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 nodes to spend computational resources to puzzle solving.

Several other solutions exist that use *social networks*[17][18] or *game theory*[19]. 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[20] malicious nodes collude with the objective of deceiving 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:

- that 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 influencing over other nodes' routing table, difficulting the capacity 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.[20] 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, as a result it 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 it 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 entries count in the 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 node 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 to 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 passes 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 attacks 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 doesn't allow the node to choose a node identifier freely. Therefore, this solution is only efficient in situations where a 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[21], public-key technology[22] 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 an 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[23][24] 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 Public Key Infrastructure (PKI) must manage the whole lifecycle of each key pair[25]. 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 computationally restricted devices. So it 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's private keys will also be known 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's 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's 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**.

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

### 2.3.1 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[26][27] 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 systems 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 CA to 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;

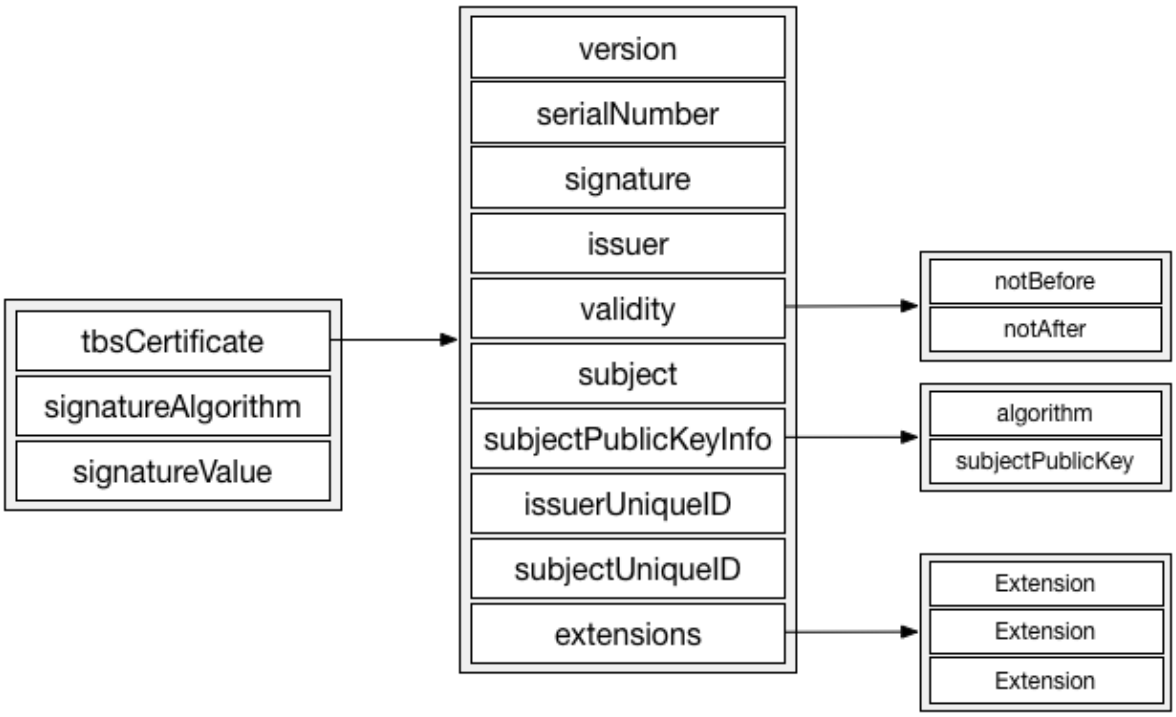



Figure 2.1: X.509 version 3 certificate structure

- **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;

  **revokedCertificates**: the list of revoked certificates. 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;

```
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

- **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)*[28] 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 has 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[29]:

- 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*[30] 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's identity through the signing of their public keys.

Pretty Good Privacy (PGP) [31] 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.

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;

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

- *Marginal*: the key ring owner only trust marginally the public key owner to sing other public keys;
- *None*: the key ring owner doens't trust 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 on 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[32] 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 consists of a linked list of blocks. Each of these blocks contain several transaction 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 [33].

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

- **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 Block 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.

*Merkle Trees* are binary trees containing cryptographic hashes built 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.

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.

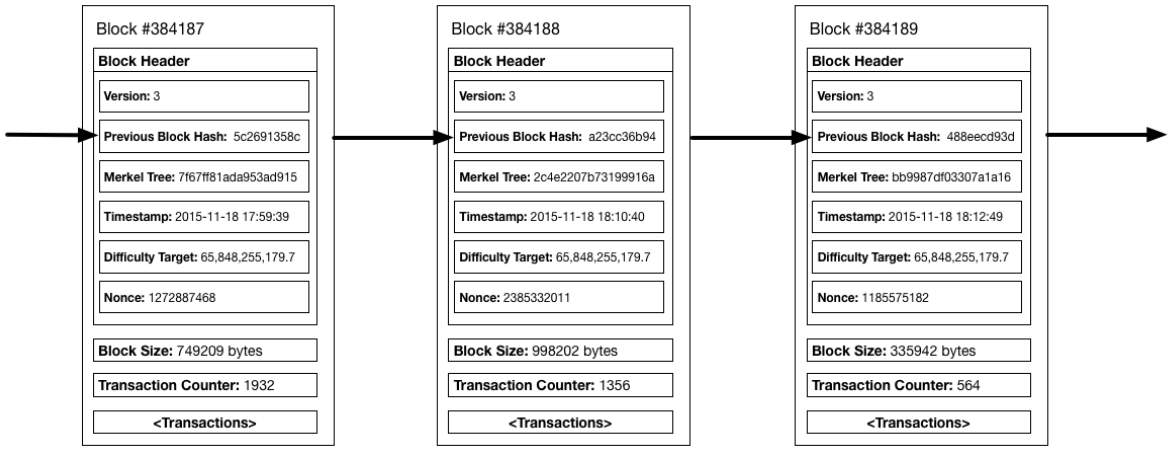



Figure 2.3: Blockchain structure and blocks content.

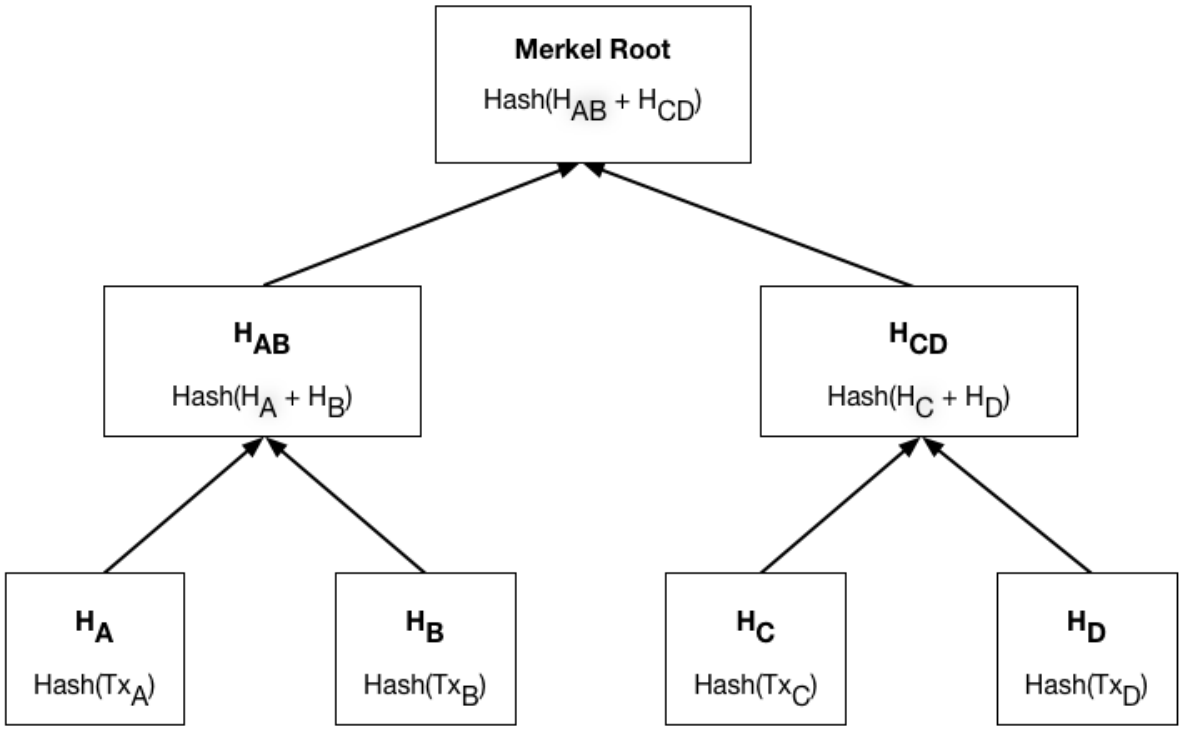



Figure 2.4: Merkle tree example

### 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.

### 2.4.5 Smart Contracts

The concept of Smart Contracts[3][34] 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]https://namecoin.info/
[3]"The Idea of Smart Contracts" - http://szabo.best.vwh.net/idea.html

### 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;
- **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).

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

- **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.

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

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

```
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.

# Chapter 3

# Architecture

This chapter describes the overall architecture of IDChain and outlines its main functional components. The main design goal 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 in more detail the architecture of the solution and discuss the main design choices.

## 3.1 Requirements

This thesis addresses the problem of building DHT-based systems in a secure fashion, mainly considering the problematics of close participation in the DHT, Sybil attacks mitigation and secure node communication.

This problem will be tackled by proposing two different solutions, a centralized and a decentralized mechanism, which will allow to control nodes' access to the DHT system.

Our overall goal is to provide a better decentralized solution and minimize trust between the intervenients.

This thesis is especially focused in building the mechanism to be used in the Global Registry component of the reThink architecture, but we want to decouple the DHT system, in order for it to be possible to use the DHT independently, enable developers to build other DHT-based systems. Therefore, the IDChain system should provide the follow functional requisites:

- Close the DHT participation, allowing only authorized nodes to join the system (federated or private system), therefore providing integrity and authentication to the DHT;
- Maintain a decentralized system that doesn't rely on a central entity or server;
- Easily add participants to the DHT;
- Deal with key compromise, by allowing to revoke nodes' certificates.

Moreover, our system must fulfill the following non-functional requirements:

- *Scalability*: The system must work well given a large number of nodes and scale easily;
- *No single point of failure*: the DHT system should work under a variable churn rate, i.e any node can be shutdown or disconnected from the network, and the system remains operational;
- *Portability*: the security mechanism used should be portable and language-agnostic, in order to be possible to integrate the IDChain in other DHT-based systems;
- *Developer usability*: the DHT system should have a simple API (get/put functions) that enables different applications to be build on top;
- *Easy deployment and management*: the system should be fairly easy to deploy and manage. An easy to use User Interface (UI) should be provided to manage all the aspects of the system.

## 3.2 Overview

Our proposal will consist of three different architectures:

- Vanilla DHT - a DHT system without any kind of peer connection security;
- DHT with CA mechanism - a DHT system where the peer connectivity is done through TLS, using a usual X509 PKI with CA infrastructure;
- DHT with IDChain mechanism - a DHT system which also uses TLS for peer connectivity, but uses certificates managed with help of a blockchain smart contract.

These three different architectures will be built not only to provide a classic approach to peer connectivity security (in case of CA mechanism), but also for evaluation purposes, mainly in terms of write/read performance of the DHT. But nevertheless, the main focus of this thesis is building the IDChain mechanism, which is the most innovative of all the presented architectures.

In Section 3.3 and Section 3.4 an overview of the vanilla and CA mechanism is given, respectively. In Section 3.5 a detailed overview of the IDChain architecture is given, including an in-depth description of each sub-component that composes the architecture.

## 3.3 Vanilla DHT

The vanilla implementation DHT represents the simpler architectural model of the presented solutions. The architecture of this solution is presented in Figure 3.1.

The DHT nodes will implement a simple API with a *put* and *get* functionality to write and read values, respectively, from the DHT.

This API is used by developers that wish to build applications on top of the DHT system. The DHT node can be used and deployed as part of applications, since it will be required as a library by the

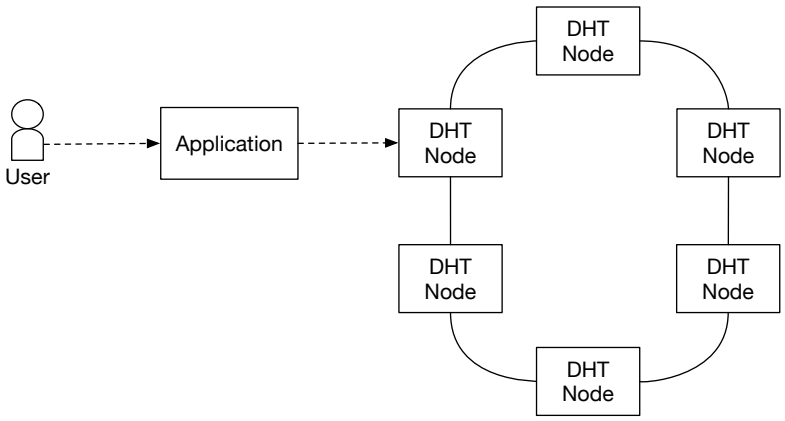



Figure 3.1: Overview of vanilla DHT architecture.

application code, which then calls directly the DHT put/get functions. The DHT nodes will communicate through an insecure channel using TCP or UDP, which are the most commonly used protocols by public facing DHT systems – for instance, UDP is used by the Mainline DHT of BitTorrent.

The node bootstrap process is done by knowing at least one of the nodes already in the DHT, and by connecting to them. This architecture lays down the basis for the other solutions' architectures, which will have the same DHT client with the put/get API but will use a secure communication protocol between the DHT nodes.

## 3.4 DHT with CA mechanism

The previous solution does not provide any kind of secure communication between nodes, opening way to a multitude of different kind of attacks, like Man-in-the-middle (MITM) attacks. Also it doesn't provide any mechanism to close the participation in the DHT (assuming the DHT nodes are Internet facing servers).

The classic approach to providing a secure communication channel between nodes is by using the Transport Layer Security (TLS) protocol coupled with a X509 PKI infrastructure. This approach is summarized in Figure 3.2.

This strategy is a good fit for a federated model since, usually, there is a consensus between the enterprises (in the Global Registry case, a Service Provider) participating in the system, and therefore it is possible to establish a CA which will issue the certificates of all the SP nodes.

This solution could be even more robust if we create a two-tier architecture, as shown in Figure 3.3.

In this hierarchy there is an offline Root CA which issues for each SP an intermediate CA certificate. These intermediate CAs are maintained online (for CRL access) and issue certificates for each DHT node controlled by the respective SP node.

In each node certificate the node identifier, IP address and/or domain should be registered.

Since we will be using TLS mutual-authentication it is required that the issued certificates could be used as server certificates and client certificates.

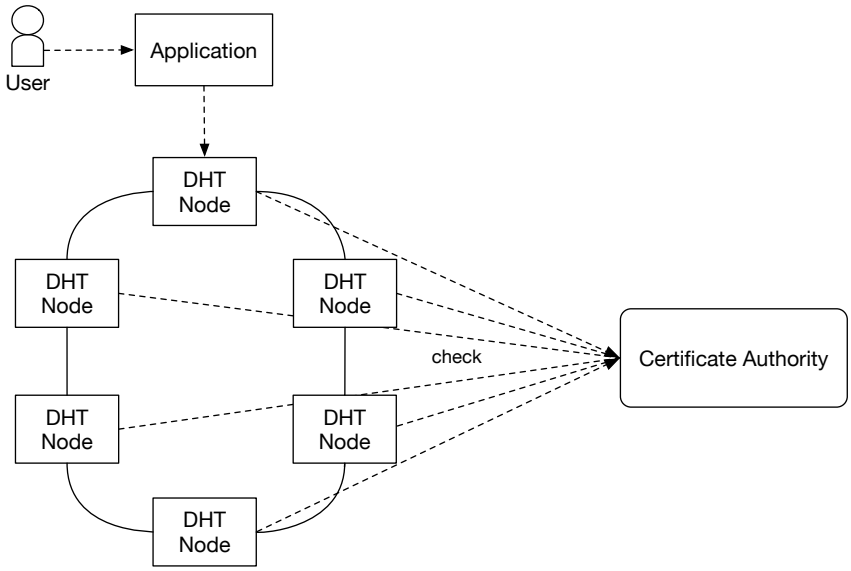



Figure 3.2: Overview of DHT with CA architecture.

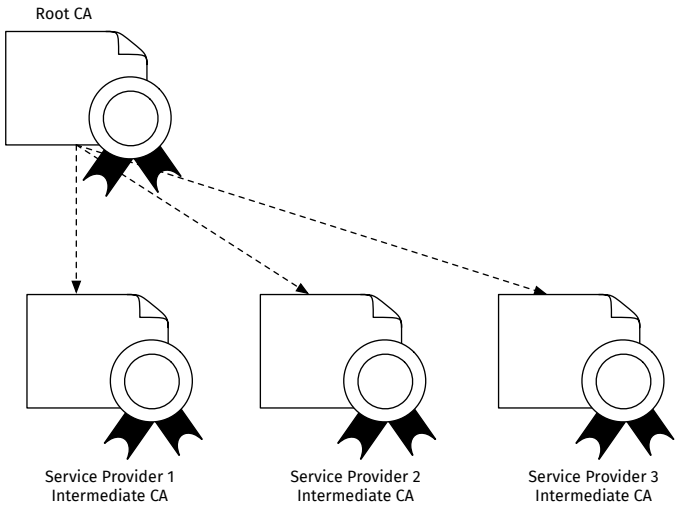



Figure 3.3: Two-tier Certificate Authority hierarchy

The node bootstrap process is the same as the vanilla solution, but when establishing the TLS connection between the nodes, it is necessary to verify during the TLS handshake that the certificates are signed by one of the valid intermediate CA's, and check if the contacting node identifier is equal to the one registered in the received certificate. If any of these two validations fail, the connection is closed. This verification is done both ways: the connection server checks if the client certificate is valid according to these specifications, and the connection client verifies if the server certificate is also valid.

This verifications are done every time that a TLS connection is established, i.e when a message is sent between nodes. This could incur in a performance decrease, since every time a message is sent these verifications are done when establishing the TLS connections. A possible solution to mitigate this performance decrease is to use TLS Session Resumption[35] mechanism that allow a TLS connection server to resume sessions, avoiding at the same time, keeping session state per-client.

## 3.5 DHT with IDChain mechanism

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 3.4 an high-level overview of the architecture is shown. In Section 3.6 and 3.7 an in-depth description of each individual component is given.

As can be seen in Figure 3.4 the two main components are the DHT and the DPKI.

The DPKI can be subdivided in three main components:

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

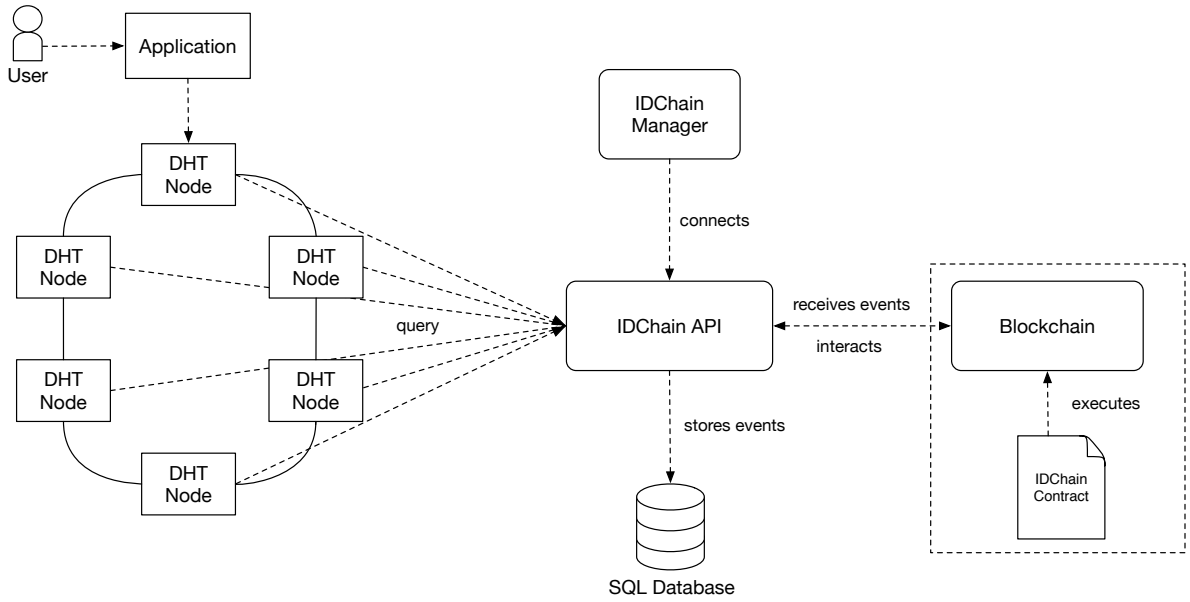



Figure 3.4: Overview of solution architecture.

In the scope of the current reTHINK project architecture, these components will only be used in a federated model, i.e the nodes in the DHT will all belong to the Service Providers (SP). Therefore it is 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 example, 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 it isn't a totally automated system and still need some human intervention, mainly when accepting and signing Certificate Signing Requests (CSRs).

## 3.6 Distributed Hash Table

The DHT design and implementation will be performed by another reTHINK project member, so there will only be a focus in the security of the DHT.

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.

Consequently, the peer should also check if there is a correspondence between the certificate fingerprint and the peer identifier registered in the blockchain.

## 3.7 Decentralized Public Key Infrastructure

Establishing 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 has access to the root 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's 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.

### 3.7.1 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 are going to create has the main objective of associating a 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 guaranteeing 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 create a mechanism of defense against Sybil attacks. The starting point are the conclusions drawn by Douceur[12] 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

With the advent of the blockchain technologies and smart contracts, it is possible to build entities and organizations like the DAO [36], 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 these entities could be defined by a smart contract deployed in a blockchain.

This way, leveraging these technologies, it 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 of the smart contract is the *Certificate* structure and fields. In Figure 3.5 are depicted the necessary fields in this structure.

Note that it is not necessary to save the full certificate metadata in the blockchain: the certificates are already exchanged in the TLS handshake between the nodes.

It could be possible to save all the certificate metadata or even the full encoded certificate for searchability and navigability purposes, but blockchains are not appropriate for storing large quantities of data.

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

Not only blockchains have a maximum amount 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 storing the full certificate in a P2P filesystem like InterPlanetary File System (IPFS), and saving the address hash of the certificate in the *Certificate* structure.

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

Figure 3.5: Smart contract *Certificate* structure fields.

Since we are trying to build this DPKI under a federated model, the trust mechanism must be defined for the system. Hence we are building the trust mechanism relying on a web of trust model built on top of the blockchain.

As is shown in Figure 3.6, a newly created entity to be accepted should be vouched for 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 provide an 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 3.7 are defined the necessary fields. It 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* field) are stored in this entity, because it will be necessary to constantly calculate the state of each entity. If we consider a graph representation (usual in the web of trust model), these fields encode the inbound arrows and outbound connections of each entity.

Another important aspect to take into account when building this smart contract is the bootstrapping process. In order for the web of trust mechanism to work it is necessary to define the initial entities which are trusted. As we are in a 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 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 provide are:

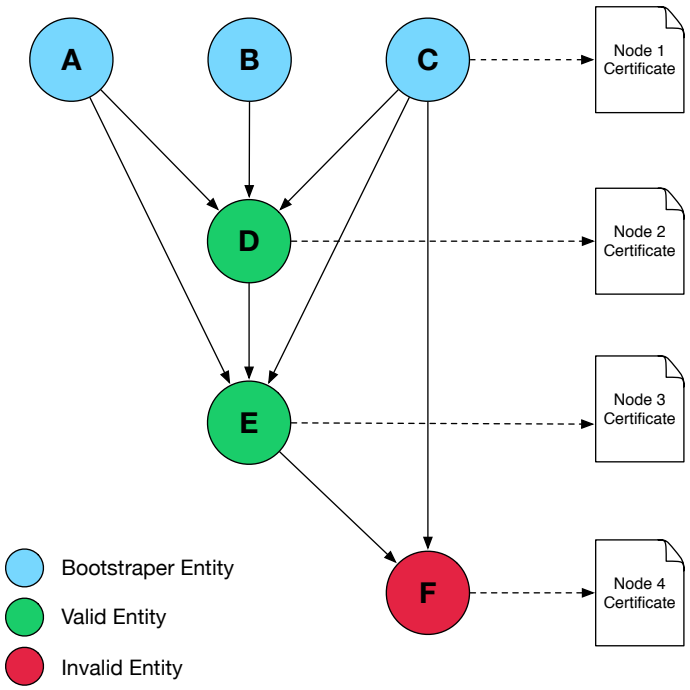



Figure 3.6: Web of trust mechanism.

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

Figure 3.7: Smart contract *Entity* structure fields.

- **Register a new entity** - this function should initialize 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, it 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

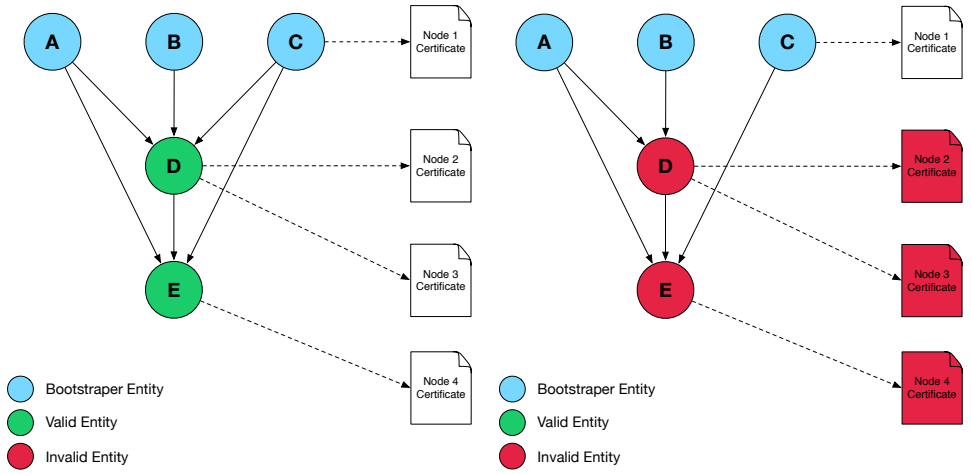
In some aspects the web of trust mechanism that we are going to implement is different from the more well-known web of trust systems.

One of these aspects, as shown in Figure 3.8(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 in a federated model, we may want for each SP cto have multiple nodes, in order for the DHT to be scalable.

The valid state of each certificate is determined by its entity state: if an entity has the minimum number of vouches required by the system, then the certificate created by the entity is considered valid. This is depicted in Figure 3.8(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 it 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 3.8(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. It is worth mentioning that the certificates of *Entity D* and *Entity E*, as mentioned before, are also considered invalid.




(a) Scenario A (b) Scenario B

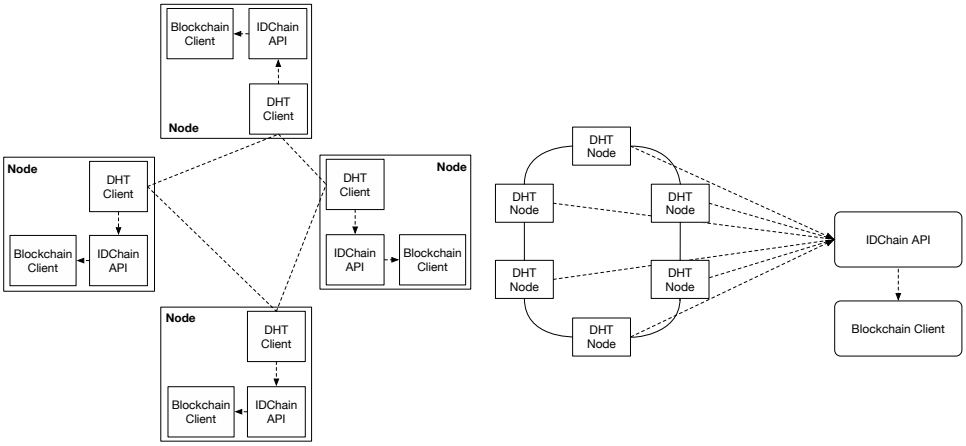
Figure 3.8: Entity invalidation scenario, with trust chain verifications.

### 3.7.2 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 3.9(a), or a multiple API instances, one for each node,

running in the same server, shown in Figure 3.9(b).




(a) API Multi-deploy (b) API Single-deploy

Figure 3.9: IDChain API deployment strategies.

The API will have the classic components usually associated with a RESTful service: every endpoint returns JSON documents and it 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 of building this API for accessing the IDChain functionalities, is that allows to build applications and services that easily interact with it through Hyper Text Transfer Protocol (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 3.7.3.

The IDChain API will be backed up by a relational database (using Structured Query Language (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.

Using a relational database allows for the structuring of all the transactions and data associated with the system, which enables more powerful queries in a simpler way.

For instance, obtaining all the certificates associated with an entity or getting the blockchain transaction where a specific revocation was made.

These kind of associations and structure are 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's *events*. It is possible to trigger events in smart contracts that when 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. Hence we can mirror the transaction information related with our smart contract to the relational database.

| 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

### 3.7.3 IDChain Management Application

In addiction we will build an application that allows to do all the operations related with the IDChain system. It will be a web application that will interact directly with the IDChain API.

The main functionalities will revolve around entity and certificate management: viewing entities vouches, creating new node certificates, viewing the transactions related with each entity, etc.

The main inspiration for this system are web-based *blockchain explorers*[2] that exist for blockchains like Ethereum and Bitcoin.

The web application will be built using a Single Page Application (SPA) architecture, leveraging browser Javascript frameworks. This will enable us to recreate a desktop application feel, keeping at the same time the web application's convenience and ease of access.

[2]https://blockexplorer.com/

## 3.8 Overlay network processes

### 3.8.1 Node bootstrap and registration

In order to participate in the DHT, the SP who wishes to deploy the node must first create a blockchain account/wallet, and then initialize his entity (one for each account) in the IDChain system.

The entity should then be validated by the other entities vouches.

Next it 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.

### 3.8.2 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 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.

It will be necessary to add an extra-step to the TLS handshake to verify the validity of the exchanged certificates. As depicted in Figure 3.10 is necessary to verify the following:

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

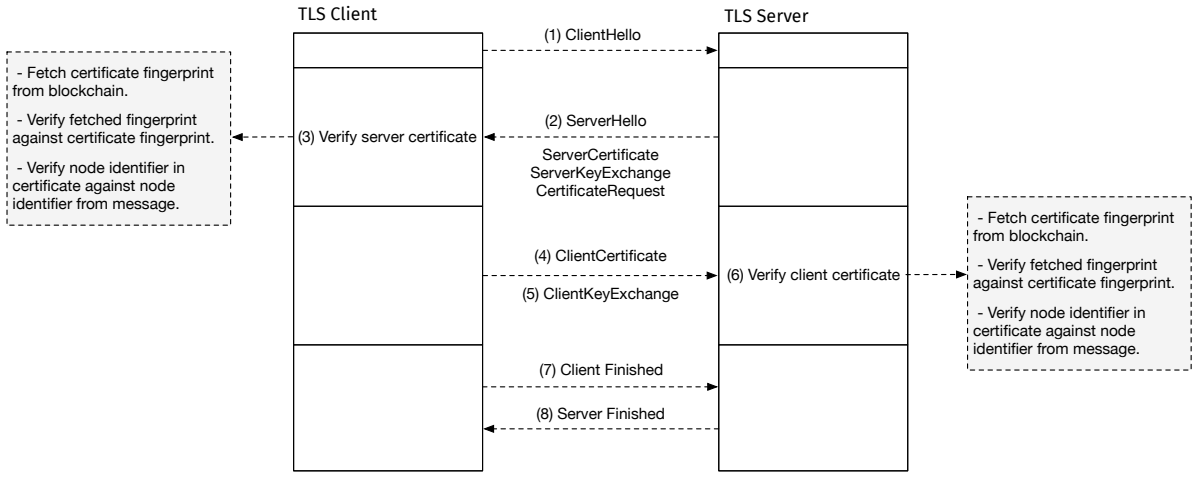



Figure 3.10: Additional steps added in TLS handshake verification.

### 3.8.3 Eclipse and Sybil attacks defense

The mitigation of possible Sybil and Eclipse attacks leverages 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 launching 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.

## 3.9 Comparison with a CA

Using a blockchain to build a PKI, it 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, it 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, if we want to deploy a private blockchain network.

# Chapter 4

# Implementation

This chapter addresses the main decisions adopted regarding the implementation of the vanilla, CA-based and IDChain systems. Therefore, the following sections cover the technologies that were used in the development process and the implementation details of each component.

## 4.1 Adopted Technologies

In this section we will present the chosen technologies to implement the proposed work for each component of the architecture.

### 4.1.1 DHT

In a first instance, the chosen library to implement the DHT node was *TomP2P*[1] an implementation of Kademlia in Java. The use of TomP2P was already decided before-hand, since the Global Registry component in the reThink project, was already partially implemented using it. This restricted the implementation options from the get-go, since it wasn't possible to compare different DHT implementations and check which one could be more easily extended.

We tried to modify the TomP2P source code, and implement TLS connection support. This endeavor didn't succeed, mainly since the TomP2P code is tightly coupled to TCP and UDP connections.

Since TomP2P uses *Netty*[2], a Java NIO client-server Non Blocking IO (NIO) framework, we tried to to change Netty instance calls to TLS, a try which revealed infeasible.

Therefore we decided to implement the mechanisms using another DHT system. Several DHT implementations exist in different languages, but it was necessary to pin down an implementation that had a modular architecture and could be easily extended.

The implementation that revealed to be the most modular was the one of *Kad*[3] library, for *Node.js*[4] and based on Kademlia. This implementation allows to easily extend the base DHT with custom trans-

[1] http://tomp2p.net/
[2] https://netty.io/
[3] https://kadtools.github.io
[4] https://nodejs.org/

ports, middleware, storage layers and message processors, which enabled us to build a custom transport with TLS mutual-authentication and custom verification rules.

The same DHT client was used for the three different mechanisms: vanilla, CA-based and IDChain. It is possible to switch between the three different mechanisms by declaring in the configuration file which one we want to use.

### 4.1.2 Blockchain

The blockchain that we used was Ethereum. There are a number of other blockchains currently available that enable developers to build applications on top of it using smart contracts, like Hyperledger Fabric[37] or R3 Corda[38], but these implementations have more specific use-cases, enterprise and finances respectively.

A quick analysis between these three platforms shows significant architectural differences, in such a way that we can even consider an higher-level classification for these three platforms, since not all of them show the main characteristics of a blockchain (taking in consideration the Bitcoin reference paper). A broader definition that encapsulates these three platforms is Distributed Ledger Technology (DLT).

The Hyperledger Fabric platform has a modular approach to blockchains. It's a private blockchain system and is permissioned, i.e doesn't allow unknown entities to participate in the system. To participate in the system the members of a Hyperledger Fabric system, must enroll through a Membership Service Provider (MSP). This allows different architectural approaches in the system, because it is not strictly necessary to have a Proof of Work (POW) mechanism, like in Ethereum or Bitcoin (which may be considered open networks), and it is possible to use different consensus mechanisms in the network. Also, there isn't any built-in currency like in Bitcoin or Ethereum.

The R3 Corda system is considered a DLT for financial institutions. It is also permissioned and private and uses a concept of notary nodes to validate uniqueness of transactions. Like Fabric, it also allows a more fine-grained control of the consensus mechanism. Data is shared on a need-to-know basis, there is no global broadcast of all the transactions in the system.

Finally, the Ethereum platform is the more generic blockchain platform of the ones referenced. It is permission-less, and can be public or private. All the participants have to achieve consensus over all the transactions of the system, using the POW consensus and leveraging the built-in currency (ether).

Our choice was Ethereum due to a number of reasons:

- Permissionless - even though we want to build our mechanism around a federated/enterprise model, this proof of concept may be easily adapted to be used as a global system. Since the Ethereum network is public, and we could also deploy private instances, it is the right fit.
- Developer ecosystem - right now, Ethereum has the biggest developer community, with a increasing number of development tools to aid the development and deploy process of smart contracts.

Despite the fact that Ethereum network is currently the system with the better developer ecosystem, there are still several shortcomings when developing against it. In Section 4.4 we detail some of the difficulties and shortcomings of the Ethereum smart contracts we encountered during development.

### 4.1.3 IDChain API

#### Application Server

The IDChain API was built using the Node.js Javascript runtimes, and the following frameworks and libraries:

- *Hapi*[5] - web framework for building web applications, RESTful APIs and services;
- *web3.js*[6] - the Ethereum compatible Javascript API;
- *Sequelize*[7] - a Object-Relational Mapping (ORM) for Node.js which support several SQL dialects.

The decision of using Node.js to develop the HTTP API is mainly related with the web3.js library. At the time of development was the most mature library to interface with the Ethereum client. The Ruby, Java and .Net implementations were still in early stages of development.

The Hapi framework was chosen because allows to build APIs encapsulating endpoints around a plugin system. For example, it is possible to create a plugin that encapsulates all the endpoints that are related to the certificates' domain. This allow us to compose the API using tiny modules that contain the business logic of each domain in the application, which later could be decoupled and deployed separately. Also Hapi already has a couple of modules built-in for dealing with input and response validation, error handling, session caching, logging, etc.

We decided to use an ORM to interact with the database due to the following reasons:

- Allows to easily declare the database schema of the different API entities;
- Leads to a huge reduction in code, since it eliminates the need for repetitive SQL queries;
- Facilitates navigation of entities' relationships;
- Transaction management and isolation;
- Independent of the SQL database or dialect used, allowing the use of any SQL database or dialect.

We decided to use the Sequelize ORM, since it is the most mature and fully-fledged SQL ORM for Node.js.

#### Database

The database we decided to use with the IDChain API was *PostgreSQL*[8].

The decision to use SQL comes from the necessity of predefining a schema and normalize the processed data from the blockchain.

This allowed us to easily establish relationships between the entities in the system (transactions, certificates, etc), which in turn allow us to query the system in a more efficient and structured way.

[5] https://hapijs.com/
[6] https://github.com/ethereum/web3.js/
[7] https://sequelizejs.com
[8] https://www.postgresql.org/

### 4.1.4 Management Application

The IDChain Management application was built as a web SPA. We used a common frontend stack to build this application: *React*[9] for building the user interface and *Redux*[10] for application state management.

By building the application using a SPA architecture, we are able to provide end-users a much-improved experience, mainly in navigation through the app, since no full page requests are required. The SPA architecture also takes advantage of the already existing IDChain API, by directly accessing it to obtain the data and control all the aspects of the system.

## 4.2 Vanilla system

The vanilla DHT we present in this section, is the barebone client for the other implementations in the next sections.

As mentioned before, the DHT which we implemented is based on the Kad DHT library. Kad provides a very thin basis for building DHT-based applications with sane defaults, also providing at the same time an extreme level of extensability and customization.

The DHT was built using Kad, which has an 160-bit address space and the nodes' communication is done through JSON-RPC using the HTTP protocol.

In order to launch every DHT client it is necessary to provide a configuration file.

```
{
  security: {
    mechanism: 'none'
  },
  nodeID: '3b78a1ece2016370bc5e4cf56bf5ecb39120e549',
  hostname: 'idchain2.com',
  port: 8080,
  seed: {
    nodeID: '0c1ddbaf332a6b247855dbbbf8d243b3f6f3036f',
    hostname: 'idchain1.com',
    port: 8080
  },
  storage: 'data/node.db'
}
```

Listing 4.1: Example DHT client configuration file.

A configuration file example is shown in Listing 4.1. The fields *nodeID*, *hostname* and *port* correspond to the node contact information in the DHT, and more specifically, in the case of the *nodeID* field,

[9] https://facebook.github.io/react/
[10] https://http://redux.js.org/

the node identifier in the DHT network. The *storage* field is the folder where the DHT client will save the node data (key, values and routing table information). This points to a directory, because the Kad library by default uses LevelDB [11] - an on-disk key-value store - for persistence. In order to enter the DHT network it is necessary to have another node's contact information, which is defined in the *seed* field. We also defined the *security* field as an extension of the default configuration process of the Kad library. This field contains all the options and information to deploy the chosen security mechanism. In the case of the vanilla DHT, the sub-field *mechanism* should be set to 'none'. The other possible values are 'ca' and 'blockchain', for the CA-based mechanism and IDChain mechanism, respectively.

The API of our DHT client is the following:

- *start()* - start the DHT node by connecting to a node in the DHT network, which is specified in the seed field of the configuration file;
- *stop()* - stop the DHT node;
- *get(key)* - obtain the value associated with the specified key;
- *put(key, value)* - store the specified value with the specified key;

It is also important to mentions the operations that are performed underneath in the Kad module, when one of those API functions are called.

When executing the *start* function the given contact is inserted into the routing table, and performs a node lookup for its node address. This lookup will return a set of *K* nodes closest to the given key, where *K* is the number of contacts held in a bucket. Then it refreshes all buckets further than its closest neighbor, i.e, will perform a node lookup for a random number in any bucket's range, which will be in the occupied bucket with the lowest index.

The *get* function will perform a node lookup for the key that we want to fetch, and therefore build a list of *K* closest nodes. When during the search a value is returned by one of the nodes, the search is abandoned and the value returned. If no value is found, the *K* closest contacts are returned. The Kad module also requires that when the search is well succeeded, the value found must be stored at the closest node that during the search didn't return the value.

Finally, the *put* function will perform a node lookup for the given key, therefore collecting the *K* closest nodes to the given key. Then it will send a store message to each of them.

Since we are building this DHT system using Node.js, in order to facilitate the access to our DHT client and easily build applications on top of it, is possible to obtain and require our DHT client as a Node Package Manager (NPM) module.

## 4.3 CA-based system

The next solution that we implemented uses the HTTPS protocol, for node communication.

[11] http://leveldb.org/

This enables us to provide privacy — since the data transmitted is encrypted — authentication to the communicating nodes and ensures the integrity of the transmitted messages.

In the typical HTTPS/TLS setup, as in web browsers, only the identity of the server is proved using X.509 certificates. Nevertheless, it is also possible to provide client-to-server authentication, by requiring that the client provide a valid certificate.

In our implementation it is necessary to provide mutual authentication, because we need to verify if the client trying to join and perform operations in the DHT network is in fact authorized to do so. It is also necessary to verify the server identity, because we want to be sure that we are bootstraping or exchanging messages with a node belonging to the DHT network.

We implemented a custom transport adapter on the Kad library, based on the HTTPS transport already built-in in Kad. This transport is based on the HTTPS module provided by Node.js base libraries, more precisely the *https.Server* and *https.Agent* classes.

When configuring a *https.Server* instance we enable by default two options: *requestCert*, which will make the server request a certificate from clients that connect and attempt to verify that certificate, and, *rejectUnauthorized* which will reject any connection which is not authorized when verified against the certification path. The *https.Agent* instance also required that we enabled the *rejectUnauthorized* option, in order to verify the server certificate against the certification path.

It is also necessary to pass to *https.Server* and *https.Agent* instances the node certificate, node private key and CA certificate bundle (which includes the intermediary and root CAs certificates).

As can we see in Listing 4.2, the *security* field of the DHT client configuration file should contain the fields *nodeCert*, *nodeKey* and *cacert*, which contains the paths to load the node certificate, node private key and the certificate bundle. All of these files should be in Privacy Enhanced Mail (PEM) [39] format. The loaded certificate should also contain the X509v3 Extension *X509v3 Extended Key Usage* with the values *Server authentication* and *Client authentication*.

```
security: {
  mechanism: 'ca',
  nodeCert: 'certs/aac2e18af9831fe87da4fc2acb1648c44789ff32.cert.pem',
  nodeKey: 'certs/aac2e18af9831fe87da4fc2acb1648c44789ff32.key.pem',
  cacert: 'certs/idchain-ca.cert.pem'
}
```

Listing 4.2: Example of security field in configuration file

As it is, this solution already allows to assert that only authorized nodes can connect to the DHT network. But we also want to verify that the nodes are not impersonating a different identity from the one they are allowed to use. Therefore, it is necessary to encode the node's identifier in the certificate and perform the validation during the TLS handshake.

We decided to encode this information in the X509v3 Extension *X509v3 Subject Alternative Name* as a DNS entry. For the sake of completeness, we also encoded in this extension the node host name and IP address.

In our custom transport it is necessary to add this mechanism when performing a request to a node and when receiving a request from another node.

In the client side, we add a listener to the request socket object for the *secureConnection* event. This event is emitted after the TLS handshaking process for a new connection was successfully completed. Our listener function checks if the node identifier encoded as a DNS entry in the Subject Alternative Name extension in the certificate is equal to the message sender identifier, and if the certificate was authorized when performing the certification path verification.

In the server side, we simply access the certificate when handling a new request, through the received request object and do the same verification as in the client side.

It is important to notice, that this verification takes place right after the TLS handshake process. In Chapter 3, our initial approach was to perform this verification mechanism during the handshake process, but the Node.js implementation does not provide the mechanisms to modify the handshake process. Even though the benefits of performing this verification, even if after the handshake process, surpass not doing it.

## 4.4 IDChain system

In this section we will divide the explanation of each component that composes this solution through different sub-sections.

In Section 4.4.1 we explain how we modified the nodes' communication process, in order to integrate our mechanism in the DHT.

In Section 4.4.2 we present the algorithms we used to create our smart contract and build the web of trust and nodes certification system.

In Sections 4.4.3 and 4.4.4 we discuss how we implemented the RESTful API to access the IDChain system, and present the web app we implemented to easily manage the system, respectively.

### 4.4.1 DHT mechanism

The implementation of the IDChain mechanism in the DHT, is very similar to the CA-based implementation: it is necessary to provide mutual authentication in each node communication.

As in the previous solution, we also implemented a custom transport adapter, based on the HTTPS module provided by Node.js base libraries.

In this solution we are using self-signed certificates for each node, since there isn't a CA backing up the assertions about each node certificate. Therefore to verify if the self-signed certificate is indeed valid, it is necessary that a valid entity registers the certificate fingerprint in the blockchain, under a node identifier.

The certificate fingerprint is the hash of a Distinguished Encoding Rules (DER) encoded certificate. The function used to perform the hashing of the certificate could vary. At first hand, we tried to perform the certificate hashing by using *SHA-256* hash function, but the Node.js TLS implementation hashes the

certificate using SHA-1 hash function, which forced us to use *SHA-1*.

Our custom transport adapter in this case will be significantly different of the one of the previous solution. We will also be using HTTPS/TLS mutual-authentication, but in this case the verifications will be different. The configuration file, as can be seen in Listing 4.3, comparing with the one in CA-based mechanism, has the *mechanism* field set to *blockchain*, and the field *cacert* removed.

```
security: {
  mechanism: 'blockchain',
  nodeCert: 'certs/aac2e18af9831fe87da4fc2acb1648c44789ff32.cert.pem',
  nodeKey: 'certs/aac2e18af9831fe87da4fc2acb1648c44789ff32.key.pem'
}
```

Listing 4.3: Example of security field in configuration file in blockchain mechanism

First of all, when configuring the *https.Server* and *https.Agent* instances, we enabled the *requestCert* option in *https.Server* (so the client certificate is requested), and set to *false* the *rejectUnauthorized* option in both instances. The *rejectUnauthorized* is set to *false* because in this case we don't have a certification path (bundle of CAs) to verify the nodes' certificates against. So this allows the self-signed certificates to be accepted in a first phase.

We encountered one problem when trying to disable the *rejectUnauthorized* option in *https.Server* instance. Even though we set its value to *false*, the *https.Server* instance was always rejecting the client certificate and closing the connection. Therefore, to overcome this problem, and verify the client certificate, it was necessary to implement an additional mechanism, which will detail later.

In the client side, we use a similar strategy as the CA-based mechanism: we add a listener to the request socket object for the *secureConnection* event that will perform our verifications. In this case it is necessary to first request the node certificate fingerprint from the IDChain API */peer/id* endpoint, then verify if the fingerprint of the received certificate is equal to the certificate fingerprint registered in the blockchain, and check if the peer identifier of the message sender is equal to the node identifier encoded in the certificate.

In the server side, as we said before it was necessary to add another mechanism to verify the client. Any request done by the connection client must have the following additional information encoded in the header:

- Client certificate - encoded in base64;
- Timestamp/challenge - in Unix Timestamp format;
- Timestamp/challenge signature - a client digital signature of the timestamp;

In the server side, when handling a new request, we fetch the client certificate from the header, and verify the timestamp signature contained in the header was performed by the same client. Then we also do the same verification as in the client side — fetch the certificate from the IDChain API, verify the certificate fingerprint against the one stored in the blockchain, and finally check if the message sender node identifier is equal to the node identifier stored in the client certificate.

The nodes' certificates we use in this mechanism, even though are self-signed, encode the same information we need as the previous CA-based mechanism. We encode the node identifier, IP address and host name as entries in the X509v3 Subject Alternative Name extension.

### 4.4.2 Smart contract

In this Subsection we detail the implementation of the functions that compose our smart contract.

We opted to use the language Solidity to write our contract, and used the Truffle framework to ease the development, testing and deployment of our smart contract.

One mechanism built-in in the Ethereum blockchain and that we take advantage of, are events. Events allow smart contracts to dispatch notifications to applications that are connected to the Ethereum client, and listening to these events. When an event is called in a smart contract, the arguments will be stored in the transaction's log, a data structure in the blockchain, that is associated with the address of a contract. These logs are incorporated in the Ethereum blockchain and will stay there as long as a block is accessible.

We use events extensively through our smart contract code, mainly to allow us to trigger some functions in the IDChain API. In Algorithm 1 are declared the events that are triggered in Algorithms 3-8. In Subsection 4.4.3 we go in greater detail on how we use events.

**Algorithm 1** Contract events declaration.

**event** $SignEntity(signer, target, timestamp)$
**event** $UnsignEntity(signer, target, timestamp)$
**event** $EntityStatusChange(entity, timestamp, valid)$
**event** $CreateCertificate(entity, id, timestamp, ip, peer, fingerprint, revoked?)$
**event** $RevokeCertificate(entity, timestamp, peer)$
**event** $EntityInit(entity, timestamp, name, bootstraper, valid)$

We present in Algorithm 2 the initial global state of our smart contract.

**Algorithm 2** Contract global state initialization.

$counterId \leftarrow name$ ▷ certificate identifier counter
$revocations \leftarrow emptylist$ ▷ identifier of revoked certificates
$certificates \leftarrow emptymap$ ▷ map associating a peer identifier to a certificate
$entities \leftarrow emptymap$ ▷ map associating a Ethereum account address to a entity

In Algorithm 3, is detailed how we do the entity initialization in the smart contract. In our implementation there is a one-to-one relationship between and entity and an Ethereum account, i.e. each Ethereum account is associated with only one entity. Therefore, we first check if an entity associated with the Ethereum account executing the *initEntity* is already created. If not, we proceed with creating the entity.

One peculiar aspect of our smart contract, is how we bootstrap the web of trust basis. Since we are building an universal web of trust mechanism, in which the trust foundation rests on a predefined number of trusted entities, it is necessary to distinguish these "bootstraper entities" from a normal entity. The main difference in a bootstraper entity, is that it is always valid by default, i.e it doesn't need to be achieve a minimum number of vouches by other entities.

In our smart contract *initEntity* function, we verify if the entity that is being created is one of the first *n* entities being created, where *n* is equal to the *MAXIMUM_BOOTSTRAPERS* constant value. If it is one of the bootstraper entities, then the field *bootstraper* in its entity structure will be set to *true*, else it will be set to *false*. This field will be useful when validating an entity's state, in the *checkValidity* function presented in Algorithm8.

**Algorithm 3** Create new entity function pseudo-code.

```
1: function INITENTITY(name)
2:     if !entities(msg.sender).created then
3:         entities[msg.sender].name ← name
4:         entities[msg.sender].created ← true
5:
6:         if bootstrapersCount < MAXIMUM_BOOTSTRAPERS then
7:             entities[msg.sender].bootstraper ← true
8:             entities[msg.sender].valid ← true
9:             bootstrapersCount ← bootstrapersCounts + 1
10:        else
11:            entities[msg.sender].bootstraper ← false
12:            entities[msg.sender].valid ← false
13:        end if
14:
15:         trigger event EntityInit(msg.sender, block.timestamp, name,
16:        entities[msg.sender].bootstraper, entities[msg.sender].valid)
17:    else
18:        throw
19:    end if
20: end function
```

After an entity is created it is now possible to create certificates that will be associated with this entity. In Algorithm4 we detail the function that is called when an entity wants to associate a new certificate. This function instantiates a new Certificate structure and stores the passed arguments in the respective Certificate fields. The certificate emitter also needs to send a pre-specified amount of *ether* to the specified smart contract address. It is necessary to "pay for the certificate" in order to have a monetary obstacle to creating several certificates.

It is also necessary to allow entities to revoke certificates that they have issued. The certificate revocation functionality is presented in Algorithm 5. There is one important detail in this algorithm: only the entity which issued the certificate can revoke it. We enforce this rule, by verifying if the entity calling the function is the signer of the certificate which it wants to revoke.

The main aspect of the IDChain smart contract is the possibility to build the trust between the different entities. The functions presented in Algorithms 6 and 7 allow us to build the necessary web of trust, by vouching and unvouching for entities. In order to do this, we store in each Entity structure two arrays: one that has the addresses of the entities that we vouched for (*signed* field), and one containing the address of the entities that vouched for us (*signers* field). Then the *signEntity* and *unsignEntity* only have to add or remove the respective entities address from the the *signed* and *signers* arrays.

These functions also need to call the *checkValidity* function, shown in Algorithm 8, at the end of their execution.

The *checkValidity* function is what checks if an entity is considered valid after being vouched or

**Algorithm 4** New Certificate function pseudo-code.

```
1: function NEWCERTIFICATE(fingerprint, ipAddress, peerID)
2:     certificates[peerID].fingerprint ← fingerprint
3:     certificates[peerID].date ← block.timestamp
4:     certificates[peerID].id ← counterId
5:     certificates[peerID].ipAddress ← ipAddress
6:     certificates[peerID].peerID ← peerID
7:     certificates[peerID].revoked ← false
8:     certificates[peerID].signer ← msg.sender
9:
10:    insert certificates[peerID] into entities[msg.sender].certificates
11:    sendTransaction(cost, contractAddress)                ▷ pay certificate creation cost
12:
13:    trigger event CreateCertificate(msg.sender, counterId, block.timestamp,
       ipAddress, peerID, fingerprint, false)
14:
15:    counterId ← counterId + 1
16:
17:    return certificates[peerID].id
18: end function
```

**Algorithm 5** Revoke certificate function pseudo-code.

```
1: function REVOKECERTIFICATE(peerID)
2:     if certificates[peerID].signer == msg.sender and !certificates[peerID].revoked then
3:         certificates[peerID].revoked ← true
4:         insert certificates[peerID].id into revocations
5:         trigger event RevokeCertificate(msg.sender, block.timestamp, peerID)
6:     else
7:         throw
8:     end if
9: end function
```

**Algorithm 6** Sign entity function pseudo-code.

```
1: function SIGNENTITY(entity)
2:     insert msg.sender into entities[entity].signers
3:     insert entity into entities[msg.sender].signed
4:     trigger event SignEntity(msg.sender, entity, block.timestamp)
5:     CHECKVALIDITY(true, entity)
6: end function
```

**Algorithm 7** Unsign entity function pseudo-code.

```
1: function UNSIGNENTITY(entity)
2:     for i < entities[entity].signers.length do
3:         if entities[entity].signers[i] == msg.sender then
4:             delete position i from entities[entity].signers
5:             break
6:         end if
7:     end for
8:
9:     for j < entities[msg.sender].signed.length do
10:        if entities[msg.sender].signed[j] == msg.sender then
11:            delete position j from entities[msg.sender].signed
12:            break
13:        end if
14:    end for
15:
16:    trigger event UnsignEntity(msg.sender, entity, block.timestamp)
17:
18:    CHECKVALIDITY(false, entity)
19: end function
```

unvouched for. If there is a change of validity state in the entity, it is necessary to check the whole downstream trust connections of the entity.

First, we consider an entity valid if it has at least *m* signers, being *m* the MINIMUM constant in the presented algorithm. If an entity has a status change in validity, it is therefore necessary to recursively call the *checkValidity* function for each entity it has signed. We pass the argument *previousEntityState*, that represents the current state of the previous entity in the chain, which is needed to know if the entity we are currently analyzing will have a state change.

**Algorithm 8** Check entity validity function pseudo-code.

```
1: function CHECKVALIDITY(previousEntityState, entity)
2:     if entities[entity].signers.length <= MINIMUM and !previousEntityState and !entities[entity].boostraper
   then
3:         entities[entity].valid = false
4:         for i < entities[entity].signed.length do
5:             CHECKVALIDITY(false, entities[entity].signed[i])
6:         end for
7:         trigger event EntityStatusChange(entity, block.timestamp, false)
8:     end if
9:
10:    if entities[entity].signers.length >= MINIMUM−1 and previousEntityState and !entities[entity].valid
   then
11:        entities[entity].valid = true
12:        for j < entities[entity].signed.length do
13:            CHECKVALIDITY(true, entity)
14:        end for
15:        trigger event EntityStatusChange(entity, block.timestamp, false)
16:    end if
17: end function
```

### 4.4.3 API

The API we built is structurally simple: it listens to the IDChain smart contract events, and processes those events by writing the information accordingly to the database. Then it exposes the stored information using a RESTful HTTP API, which contains several endpoints.

We defined a database schema which contains different tables to each main structure of the smart contract. In Figure 4.1 we present the entity-relation model of the database, and describe each database table. It's important to notice that we defined a *Transaction* table that doesn't have any relation with the other tables. We did this as a first iteration of the system where we only want to view all the transactions related with a specific entity. We also defined the *Signatures* table where the vouch relations between the entities are stored.

Each event listener will perform a different set of operations in the database, according to the type of event. In Table 4.1, we describe the database operations that are done for each type of event.

Each of these event listeners will also create a new entry in the *Transactions* table, which will store the event information and the transaction hash associated with the triggered event.

One particular aspect that is necessary to take into consideration, is the fact that if we remove the event listeners (due to API downtime, for example) and the blockchain client keeps running and processing incoming blocks, there will be discrepancies or information missing in the API database. Even though we didn't implement it, one possible solution is to loop through all the events created by a smart contract, since event logs are permanently stored in the blockchain, and therefore create in the database the missing transactions.

### 4.4.4 Application

The IDChain Management application is built as a SPA, which means that when navigating the application the data is fetched as needed and the page updated accordingly, without page refreshes. The data is fetched from the IDChain API, using the web browser *fetch API*.

We implemented the following functionalities:

| **Event** | **Database operations** |
|---|---|
| EntityInit | Create new entry in *Entities* table |
| CreateCertificate | Create new entry in *Certificates* table. |
| SignEntity | Create new entry in *Signatures* table with the specified *target* and *source* fields. |
| UnsignEntity | Removed entry in *Signatures* table with the specified *source* and *target* fields. |
| EntityStatusChange | Update *valid* field in specified *entity* row in *Entities* table. |
| RevokeCertificate | Set *revoked* field value to *false* in the specified row in *Certificates* table. |

Table 4.1: Database operations associated with each event listener.

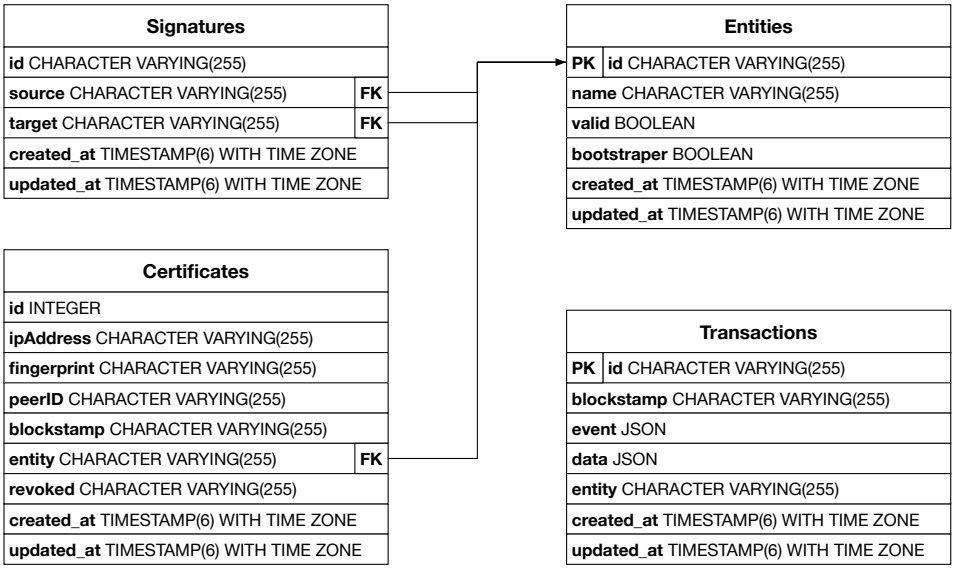



Figure 4.1: Database entity-relation model.

- **Register new certificates** — we added a form where a user can register a new node certificate, by drag-and-drop the certificate in delimited area in the page, as shown in Figure 4.2. The application automatically obtains the necessary fields from the certificate, and on submit performs the request to the IDChain API;
- **Sign/Unsign new entities** — we also add an option to allow the user to sign/unsign a specific entity in the interface;
- **View certificates registered by the entity** — the node certificates registered by the user's entity are also listed in the application, as is shown in Figure 4.3;
- **View all transactions associated with the entity** — it is possible to list all transactions the entity performed in IDChain smart contract;

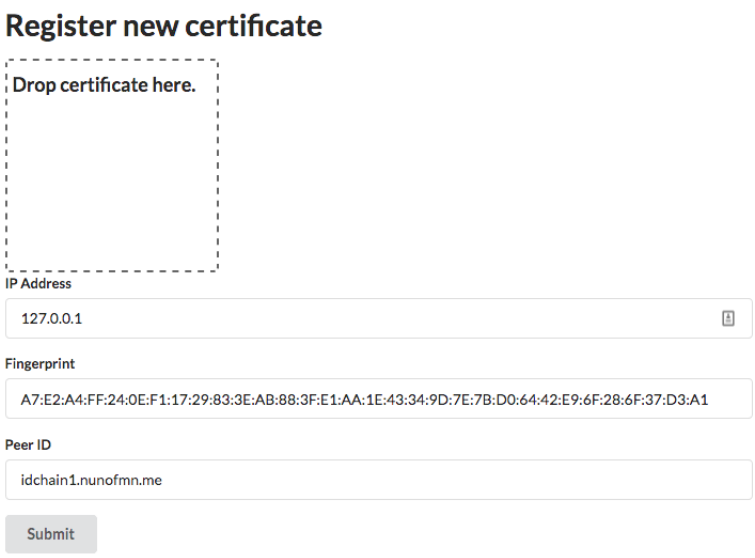



Figure 4.2: Register node certificate form.

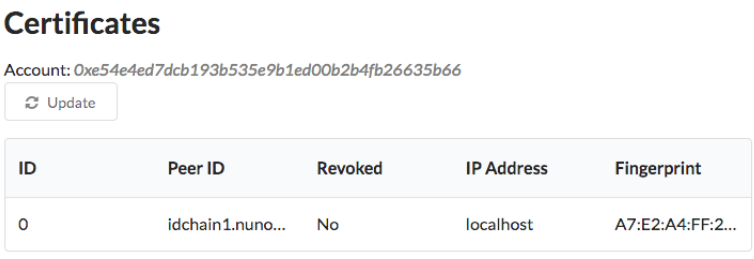



Figure 4.3: List of entity node certificates.

# Chapter 5

# Evaluation

In order to evaluate the implemented solution, several tests were done to assess protocol correctness and measure system performance. In Section 5.1 we detail our evaluation process and the overall objectives we pretend to achieve in the evaluated scenarios. Then, in Section 5.2 we describe the methods, tools and challenges we had to perform the evaluation. Lastly, in Section 5.3 we present and discuss the solutions evaluation results.

## 5.1 Evaluation Objectives and Scenarios

Our evaluation has an extended focus in the correctness of the implemented protocol. But we also perform benchmarking of the all implemented solutions, in order to compare them.

Given the requirements presented in Section 3.1, the following metrics were defined:

- *Response time for writes* – we will evaluate the system response time, as the number of write requests in the DHT increases;
- *Response time for reads* – we will also evaluate the system response time, as the number of read requests in the DHT increases;
- *Error rate* – this metric has distinct interpretations for read and write requests. In the case of the write requests, we analyze the number of nodes that stored the value, and define different error categories. In read requests, we consider that the request failed when is impossible to obtain a value.

Using these metrics, we performed load tests and functional security tests to the DHT. We performed two different types of load test: *(a)* one client only issuing write requests, (b) one client only issuing read requests. As summarized in Table 5.1, we run the tests for the different scenarios using a DHT network with 10 nodes deployed. We conducted tests for each mechanism and write/read combination, where we varied the rate from 0.1 requests/s up to 100 requests/s in write tests, and from 1 request/s to 100 requests/s in read tests. Only one node was setup to perform the requests, and each test was repeated

5 times. The tests were executed over the length of several days, in order to minimize any potential effect related to varying network traffic.

| Test # | Mechanism | Test Type | # Number of nodes | # Requests/second |
|---|---|---|---|---|
| 1 | None (HTTP) | Write | 10 | [ 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 40, 100 ] |
| 2 | CA-based (HTTPS) | Write | 10 | [ 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 40, 100 ] |
| 3 | IDChain (HTTPS) | Write | 10 | [ 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 40, 100 ] |
| 4 | None (HTTP) | Read | 10 | [ 1, 2, 5, 10, 20, 40, 100 ] |
| 5 | CA-based (HTTPS) | Read | 10 | [ 1, 2, 5, 10, 20, 40, 100 ] |
| 6 | IDChain (HTTPS) | Read | 10 | [ 1, 2, 5, 10, 20, 40, 100 ] |

Table 5.1: Test scenarios.

We also defined a metric for the smart contract evaluation:

- *Monetary cost* – we will also measure the cost of executing each smart contract function, with a variable number of entities.

This test was executed in a local machine, by calculating the *gas* cost of executing every smart contract function in the EVM.

Finally was also necessary to evaluate the DHT protocol correctness under a number of possible scenarios, from a security analysis perspective, which is done in Section 5.3.2.

## 5.2 Evaluation methodology

### 5.2.1 DHT Nodes deployment

The DHT nodes were deployed on Digital Ocean[1], distributed across several different datacenters.

We deployed 9 nodes, one node for each of the following datacenters: New York City 1 (NYC1), New York City 3 (NYC3), San Francisco 1 (SFO1), Toronto 1 (TOR1), London 1 (LON1), Frankfurt 1 (FRA1), Bengalore 1 (BLR1), Amsterdam 2 (AM2) and Amesterdam 2 (AM3). The 9th node which corresponds to the benchmark client was located in Lisbon.

The node that was defined as bootstrap node in all nodes configuration was the node located in the NYC1 datacenter.

All the nodes used a Virtual Machines (VMs) with the same specs: 1vCPU, 512 MB RAM, 20 GB SSD disk space, running Ubuntu 16.04.3 x64. The nodes used Node.js v8.2.1, PostgreSQL 9.5.8 and Ethereum testrpc client v4.1.3.

The benchmarking client was run on a server with Dual Intel Xeon E5-2640@2.00GHz CPU with a total of 32 cores, 128GB of RAM and running Debian 8.2. The versions of the software used were equal

[1] https://www.digitalocean.com/

to the Digital Ocean nodes.

We used in the DHT node configuration a k-bucket size of 5.

### 5.2.2 Tests configuration and tooling

In order to perform the load test evaluation of systems, was necessary to build a set of scripts. We built two different scripts for write and read requests. The write request script receives three parameters: DHT mechanism to use, total number of requests and demanded request rate per second. This script was ran using the following process:

1. Deploy all 9 nodes;
2. Run the script in benchmarking node with the given parameters.
3. When the script finishes, delete the all nodes cached data;
4. Repeat the process.

Is necessary to delete the node cached data between runs, in order to the tests to run in a clean state DHT, withou any data written to. The script outputs each request start/completion time and number of nodes that stored the written value in a CSV file.

The read request script receives three parameters: DHT mechanism to use, total number of requests and a list of demanded request rates per second. This script - on contrary to the write script - in each execution will perform several test iterations, increasing the demanded request rate, according to the received list of demanded requests. We choose this approach in read script because it's not necessary to clear the node data between executions, and allow us to perceive the system performance with an increasing number of requests.

In Tables 5.2 and 5.3 we present the demanded request rate and total number of requests that were done, in write script and read script executions, respectively.

| **Demanded Request Rate (req/s)** | 0.1 | 0.2 | 0.5 | 1 | 2 | 5 | 10 | 20 | 40 | 100 |
|---|---|---|---|---|---|---|---|---|---|---|
| **# of Requests per run** | 720 | 1440 | 3600 | 3600 | 3600 | 20000 | 20000 | 20000 | 20000 | 20000 |
| **Total Requests** | 3600 | 7200 | | 18000 | | | | 100000 | | |

Table 5.2: Number of write requests performed according to demanded request rate.

| **Demanded Request Rate (req/s)** | 1 | 2 | 5 | 10 | 20 | 40 | 100 |
|---|---|---|---|---|---|---|---|
| **# of Requests per run** | | | | 10000 | | | |
| **Total Requests** | | | | 50000 | | | |

Table 5.3: Number of read requests performed according to demanded request rate.

In order to write values to the DHT, we pre-generated several keys by hashing an increasing integer counter, using a SHA-1 function. The object stored under each key in the DHT, was always the same: a JSON file containing a structure similar to values that the Global Registry stores with a filesize of 827 bytes.

The certificates that were used in the test nodes were created using OpenSSL[2] and cfssl[3] command-line tools, and respect the certificate requirements that we referred in Sections 4.4 and 4.3. In the case of the CA-based mechanism tests, we used a two-tier architecture, with a root CA and a intermediate CA.

### 5.2.3 Evaluation challenges

Our benchmarking client presented performance limitations when executing our test scripts, that taken into consideration when analyzing the results obtained. Since, our DHT is based in Node.js runtime, there is limitation in the number of concurrent connections that can be done. The node running in the Digital Ocean datacenters, had by default a maximum number of open file descriptors equal to 1024. We tried to increase the value, but the limitations persisted. This causes an enormous limitation when executing the tests, because the main bottleneck was the benchmarking client.

## 5.3 Evaluation Results

In the following sections we discuss the evaluation results. The Section 5.3.1 focus in read and write load tests, and Section 5.3.3 focus in smart contract execution metrics.

### 5.3.1 Load tests

#### Write requests

The Figures 5.1– 5.5 represent the write load test evalution results.

The graph from Figure 5.1 represent the effective request rate in function of the demanded request rate, for each of the implemented solutions. Each point in the graph, illustrate the average request rate of the 5 different test executions performed, for the specific demanded request rate.

In the Figure legend the *http* value represent the vanilla DHT implementation, the *https* value represent the CA-based DHT implementation and *https-bc* represent the IDChain DHT implementation. For instance, the point (0.1, 0.1), in any of the solutions, represent the average request obtained (0.1 req/s) for 5 repetitions performed for this test, demanding a request rate of 0.1 req/s.

In Figure 5.1 is possible to verify that in the [0.1, 10] request/s demanded request rate range, all the three solutions have an approximated equal effective request rate, which means that the client is able to keep up with the demanded request rate. In the [20, 100] request/s demand request rate range, the vanilla DHT implementation outperforms the CA-based and IDChain solutions, by being able to

[2]https://www.openssl.org/
[3]https://github.com/cloudflare/cfssl

keep up more closely with the demanded request rate. The vanilla DHT implementation is able to perform $\approx$ 90 requests/second, where the CA-based and IDChain solutions become saturated at $\approx$ 17 requests/second. This shows us that the HTTP-based solution is capable of outperforming clearly the HTTPS-based solutions, at the cost of didn't provide any security aspects in the nodes communication.

The message overhead of the HTTPS handshake protocol, limits the request rate capacity of the CA-based and IDChain solutions. Is important to notice, that could be possible to increase the request rate capacity of the HTTPS-based solutions, if TLS Session Resumption and Keep-Alive connections are activated in HTTPS. We are also able to conclude that the IDChain solution is a viable solution performance-wise to the CA-based solution. It seems that increased message header overhead and the requests performed to the IDChain API, doesn't incur in a substantial cost to the effective request rate the client is able to perform, in comparison with the CA-based solution.

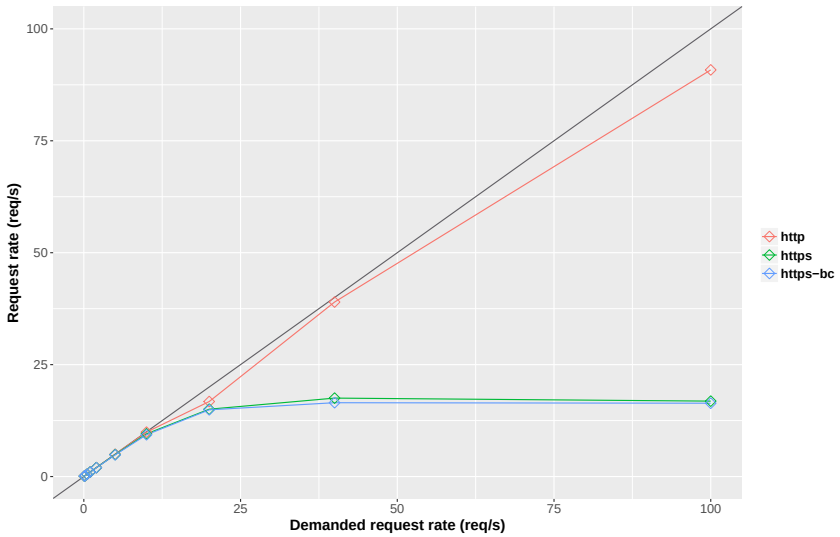



Figure 5.1: Demanded request rate.

The graph from Figure 5.2 represent the average request response time in function of the effective request rate, for each of the implemented solutions. Each point in the graph, illustrate the average request response of the 5 different test executions performed for each of the demanded request rates, in function of the effective request rate.

We can draw similar conclusions to those of the previous discussed figure: the HTTP-based implementation clearly outperforms the HTTPS-based implementations, by having, in average, $\approx$ 2 times faster response times. This results of the aforementioned added message overhead of the HTTPS handshake protocol.

The average response time of the CA and IDChain based DHT implementations remain approximately constant, only increasing when an $\approx$ 15-17 requests/second effective request rate is achieved, which correspond to a demanded request rate in the ]10, 100] requests/second range. In the case of the HTTP-based implementation, the average request response time also remained constant with an increasing load, only increasing drastically to an average response rate of 23552.88 ms when an effective request rate of 90.83 requests/second - which corresponds to a demanded request rate of 100

request/second - is achieved. This result is due to the fact that when performing that effective request rate, the test DHT client becomes saturated with inbound and outbound requests, consequently the average response time drastically increases. This only occurs in the HTTP-based implementation, because the effective request rate is much higher than in HTTPS-based solutions, which means that the bottleneck is in the client capacity of processing all inbound and outbound requests.

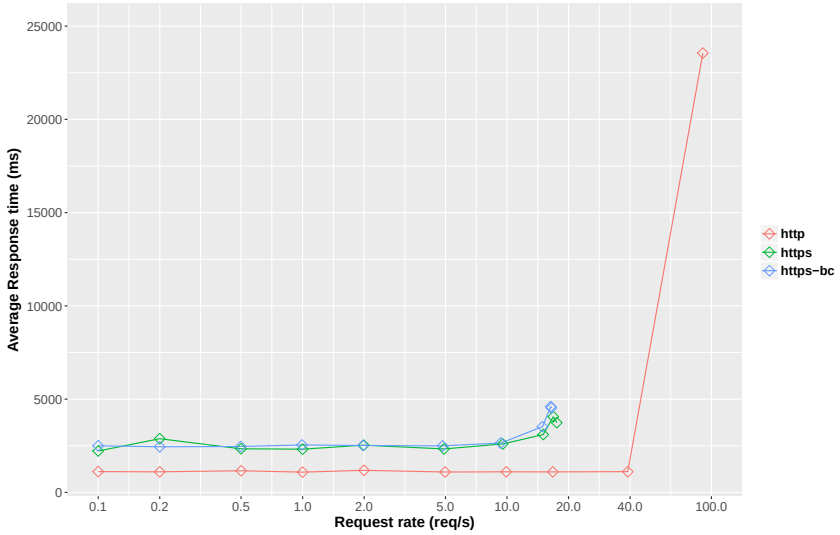



Figure 5.2: Average response time.

The graph from Figures 5.3, 5.4 and 5.5 illustrate the number of nodes that stored the value in each write request, in function of the demanded request rate for each implementation. In order to be easier to interpret the data, each plot in the graph represents a range of different number of nodes that stored the information. We represent the values in the *y-axis* as the ratio between the number of requests with the defined ”stored” variable and the total number of requests performed for the associated demanded request rate.

Is possible verify that in the HTTPS-based implementations that each value written to the DHT is almost always stored in 5 different nodes, intermittently being stored in 4 nodes. In the case of the HTTP-based implementation, in the [0.1, 40] requests/second range of demanded request rate, the obtained results are equal to those of the HTTPS-based solution. When the demanded request is increased to 100 requests/second, the number of requests in which the value is stored in 5 nodes decreases to 36.1%, and the number of failed writes (values that aren't stored in any node) increases to 63.8%. Those results in the HTTP case are closely related with the conclusions that we drawn out from Figure 5.2. Since there is a bottleneck in the client capacity of processing inbound and outbound requests, some requests will invariably timeout and the values will not be stored in any node.

### Read requests

The Figures 5.6– 5.8 represent the write load test evalution results.

As in the previous write load tests analysis, in Figure 5.1 is depicted the effective request rate in function of the demanded request rate, for each of the different solutions, where each point in the graph

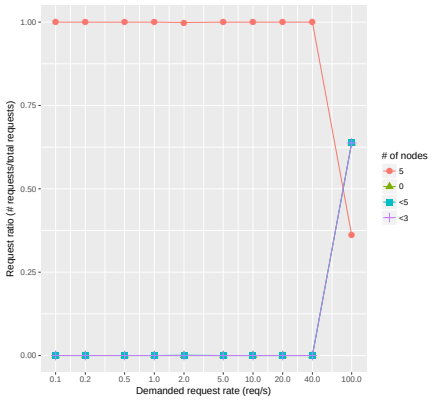



Figure 5.3: Number of nodes storing each value, in vanilla DHT.

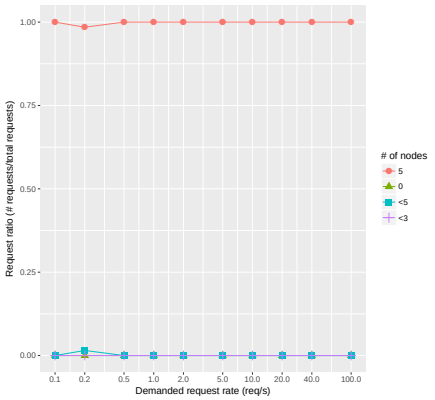



Figure 5.4: Number of nodes storing each value, in CA-based solution.

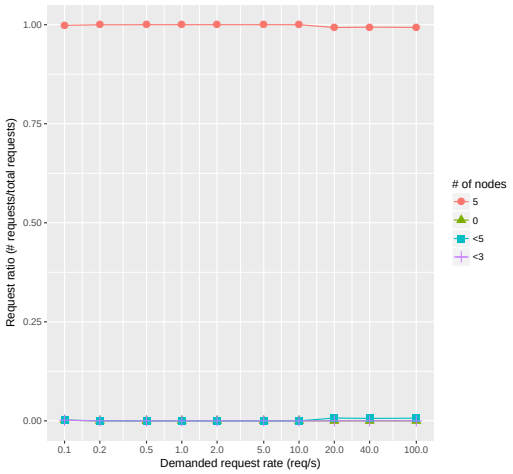



Figure 5.5: Number of nodes storing each value, in IDChain solution.

represent the average request rate of the 5 different test executions performed.

In graph from Figure 5.6 we can see that the results similar to the write tests results. In the [1, 10] requests/s demanded request rate range, is possible to verify that in all three implementations the client is able to keep up with the demanded request read. As in Figure 5.1, in [20, 100] demanded request rate range, the HTTP-based implementation outperforms the HTTPS-based implementations, being able to keep up with the demanded request rate. The vanilla DHT implementation is able to perform an effective request rate of $\approx$ 91 request/s, and the IDChain and CA-based implementations become saturated at $\approx$ 20 requests/second — results similar to those in the write requests load test.

From Figure 5.7, is possible to verify that the HTTP-based solution has, in average $\approx$ 2 times faster response times that those of the HTTPS-based solutions. Also, has stable average response times, only increasing when performing an average effective request of $\approx$ 91 requests/second (100 requests/second demanded request rate) where the average response time is of 2381.225 ms. This drastic increase — as referred in the write requests load test also — is due to the fact that since HTTP is able to perform a much higher effective request rate, the system ins unable to keep up with the processing of all inbound and outbound requests, and therefore, some requests will take longer to complete.

The HTTPS-based solutions also keep stable average response times until achieving $\approx$ 15-20 re-

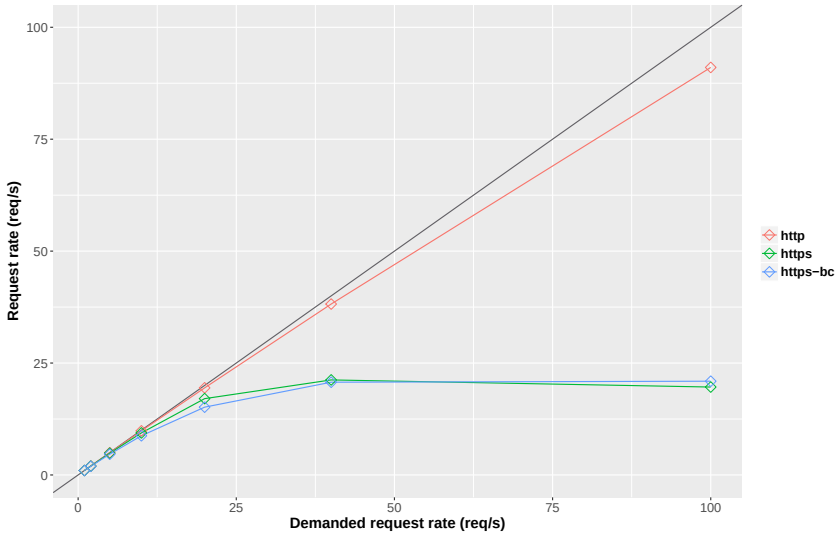



Figure 5.6: Demanded request rate.

quests/seconds effective response rate range, which is when the client starts to saturate with requests and is not possible to keep up with the demanded request rate. When the client starts to saturate, the average response time starts to increase to the 700–900 ms range.

If we compare the Figures 5.7 and 5.2, the average response times of the read requests are much lower than the write requests. This is related with the DHT protocol, which use parallel queries for searches and returns the value as soon as one node replies. In the case of write requests as is necessary to store the value in several nodes and wait for confirmation, the average response time is higher.

These results are very similar to those obtained in the write request load test also, and enable us to conclude that performance-wise the IDChain solution is a valid alternative to the CA-based solution.

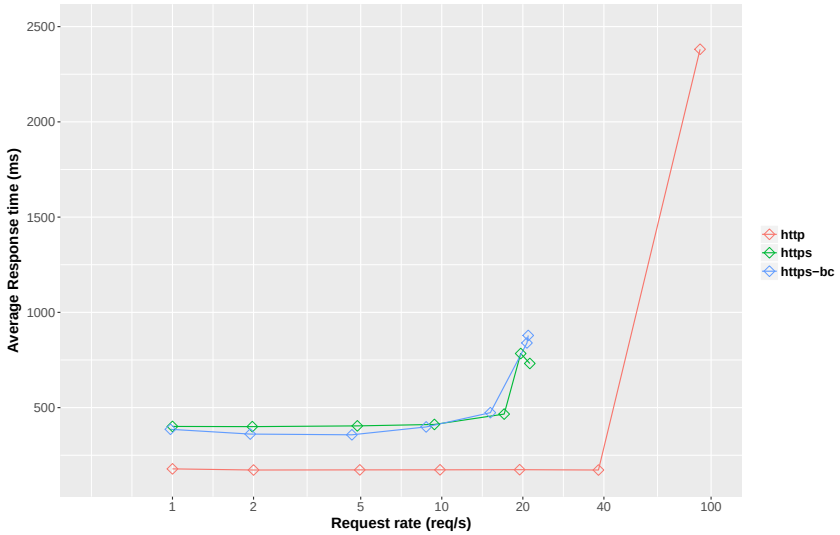



Figure 5.7: Average response time.

In Figure 5.8 is illustrated the request error ratio in function of the demanded request rate, for each of the presented solutions. We consider that a request returned an error when it's not possible to obtain the value associated with the key we are searching for.

The three implementations maintain stable error ratio between 1—2.5%, with the error ratio of the HTTP-based implementation rising to 10.4% with a demanded request rate of 100 requests/second.

The higher error ratio when achieving 100 request/second demanded request rate, in the case of HTTP was already expectable, since it is related with the aforementioned benchmark client bottleneck, and the inability to process all the inbound/outbound requests.

We were expecting the error ratio to be closer 0% in all the other cases, as an average of 1—2.5% of request errors could be considered high for designing some systems which rely on a DHT.

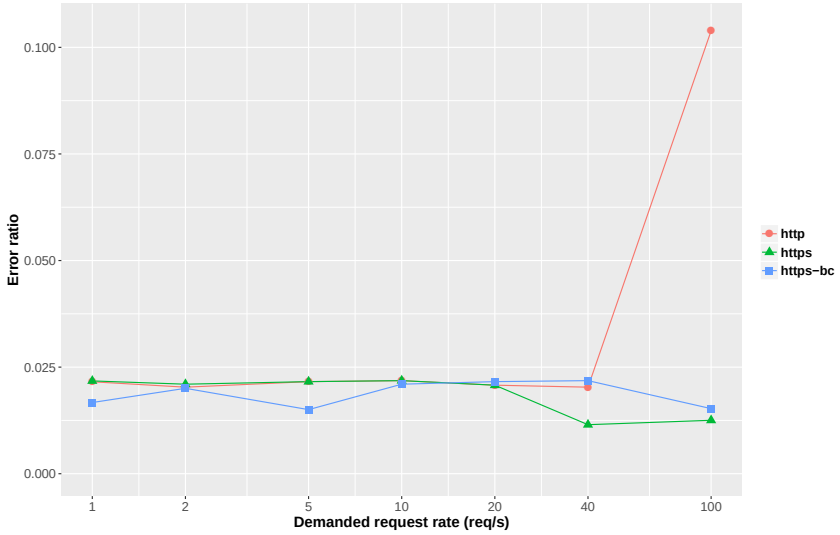



Figure 5.8: Requests error ratio.

### 5.3.2 Protocol correctness

In order to evaluate the CA-based and IDChain DHT protocol implementations, is necessary to define the threat model. Two of the main objectives of the presented solutions are to *close DHT participation* and *ensure communication privacy, integrity and authenticity*.

Taking in consideration that we are dealing with a DHT system, that are some critical DHT processes that could be potential attack vectors. We will evaluate if each one of those processes, and analyze how attack vectors are mitigated by our solutions.

A node bootstrap in the DHT, is one of the critical DHT processes which could be targeted. Each DHT node entering the network must be sure that is entering it through a trustable node, and is not being tricked into entering a DHT network with malicious nodes. Every DHT node must also ensure that every node bootstraping through them, is allowed to enter the network. This is also what enable us to provide access control to the DHT network, which take us in the direction of solving the objective of close DHT participation. Also, is necessary to ensure privacy, integrity and authenticity in the DHT network message routing, because we want to prevent any external interference in the DHT network - as for example, resorting to MITM attacks - and provide confidentiality in the data exchanged inside the DHT network. All of these aspects, are closely related to the node's communication establishment.

Is important to notice that our solutions don't pretend to deal with rogue nodes belonging to malicious organizations from the federation/consorcium using the DHT.

But since one of the objectives is *minimize trust* within the federation, we also analyze how our solutions hold up in case of a malicious organization tries to subvert the DHT protocol.

#### CA-based solution evaluation

The CA-based solution is capable of mitigate each of the aforementioned attack vectors. The TLS/HTTPS protocol with mutual authentication is able to enforce the routing messages privacy, integrity and authenticity, in the two-ways of the communication channel. Also, using X.509 certificates containing the DHT node identifiers, issued by a root or intermediate CA, we can ensure during the bootstrap process that we are effectively connecting to a DHT composed of trusted nodes.

This solution is also able to ensure that each trusted node — in case of going rogue — is not able to steal other node identifier.

One possible vulnerability of the mechanism is related with performing the extra verification of node identifier encoded in the certificate. As we are only able to perform the verification after the TLS handshake — when the *onSecure* event is called) — there is a small time frame where data could be exchanged with potential malicious nodes.

But still, this attack could only be performed by a trusted node that decides to go rogue, since the verification of the node's certificate against the certification path is performed as part of the TLS handshake process.

#### IDChain solution evaluation

As in the CA-based solution, the IDChain solution is also capable of mitigate the same attack vectors, since the solution also uses TLS/HTTPS protocol for node's communication. This solution therefore, also suffers for the same problem as the previous evaluated solution — we are only to perform the certificate peer validation after the TLS handshake process. In this case is also necessary to perform a request to the IDChain API, which slightly increases the time frame where data could be exchanged with a potential malicious node. The attack could be also performed by malicious nodes from outside the DHT network, since we are using self-signed certificates and a certification path to verify the certificates against don't exist.

This solution is not using TLS mutual authentication due to the Node.js TLS restrictions we encountered, and was necessary to add a complementary scheme in order to perform TLS clients authentication. This solution is vulnerable to one attack that could performed by a node from the DHT network, that decides to go rogue. This attack mechanism is similar to a replay attack:

1. The node that decides to go rogue performs several message exchanges with the target node, and stores the target node certificate and each 'challenge' and 'challenge signature' fields from the messages sent by the target node.

2. The rogue node now has a series of timestamps/challenges and challenge-signature fields, that could be used to impersonate the target peer identity, during a limited time frame.

This attack is only possible to be performed by a rogue node inside the DHT network, because the TLS client certificate only sends the certificate when the TLS valid handshake is performed, which means that only a trusted TLS server will receive the specified message header fields and certificate.

### 5.3.3 Smart contract execution costs

We evaluated the costs of executing each smart contract function. In order to perform the evaluation we executed each function in *testrpc* and obtained the *gas* cost of its execution. It is possible to calculate the gas cost in testrpc because it uses the same EVM as the real network client, and the total gas cost of the execution of a smart contract function comes from the sum of each low-level operation in EVM gas cost.

By default, each gas unit has a price defined in *wei* unit which is equal to $1x10^{-18}$ *ether* units. We then performed a conversion to euro currency using the conversion rate of $1ether = 286.662euros$ as of *13 October 2017 at 16 PM* in the main Ethereum network.

We calculated the cost also for different confirmation speeds in the network. It is possible to increase the amount of wei we are able to pay for each gas unit, so mining nodes give higher priority to our transactions, which speeds up the inclusion of our transactions in a valid block

In Table 5.4 are defined 3 confirmation tiers which we used to calculate transactions cost in function of confirmation speed. The data was retrieved from the ETH Gas Station[4] website.

| **Tier** | **Gas Price (Gwei)** | **Maximum Transaction Confirmation Time (minutes)** |
|---|---|---|
| SafeLow (<20 minutes) | 4 | 20 |
| Standard (<5 minutes) | 6 | 5 |
| Fast (<2 minutes) | 25 | 2 |

Table 5.4: Transaction confirmation speed tiers.

We defined two different scenarios with variable web of trust size, mainly to analyze the cost of execute recursively the *checkValidity* function in the context of the *unsignEntity* and *unsignEntity* functions. We also initiated entities with different names length, so it is possible to verify the cost difference of increasing the entities' name length. In the case of the *newCertificate* function calls we used a standard IPv4 address, a 160 bit node identifier and SHA-1 certificate fingerprint.

In the following scenarios is also considered that the minimum number of vouches necessary to a entity to become valid is equal to 3.

[4] http://ethgasstation.info/

For this evaluation the *sendTransaction* was not performed in the context of the *newCertificate* smart contract's function.

**Scenario 1**

In this scenario we performed the following operations:

a. Initiate entity 0 (initEntity function) with *name* value of *entity1*;

b. Initiate Entity 1 with *name* value of *entity10*;

c. Initiate Entity 2 with *name* value of *entity100*;

d. Initiate Entity 3 with *name* value of *entity1000*;

e. Entity 0 vouches for Entity 3 (signEntity function);

f. Entity 1 vouches for Entity 3;

g. Entity 2 vouches for Entity 3;

h. Entity 0 registers a new certificate (newCertificate function);

i. Entity 1 registers a new certificate;

j. Entity 1 revokes the previously registered certificate (revokeCertificate function);

k. Entity 0 revokes the previously registered certificate;

l. Entity 0 unvouches Entity 3 (unsignEntity function).

The cost of each operation in function of the transaction confirmation speed is depicted in Figure 5.10, the web of trust formed by perfoming those operations is represented in Figure 5.9.

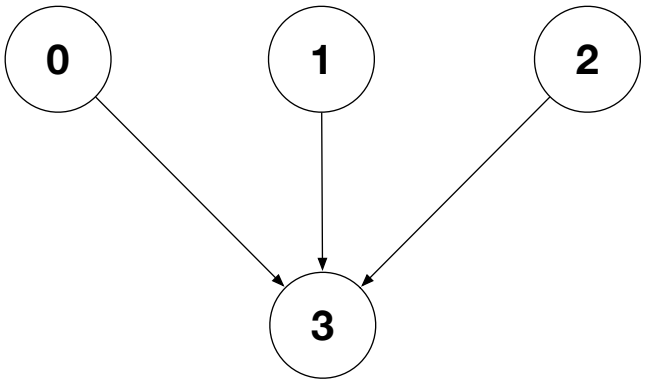



Figure 5.9: Scenario 1 web-of-trust graph.

As we can see in Figure 5.10 – assuming a "Fast" confirmation speed – is possible to verify that almost every operation we performed had a value below 1 euro, with exception of the operations using the *newCertificate* operation which execution cost goes over 3 euros. This increase of execution cost

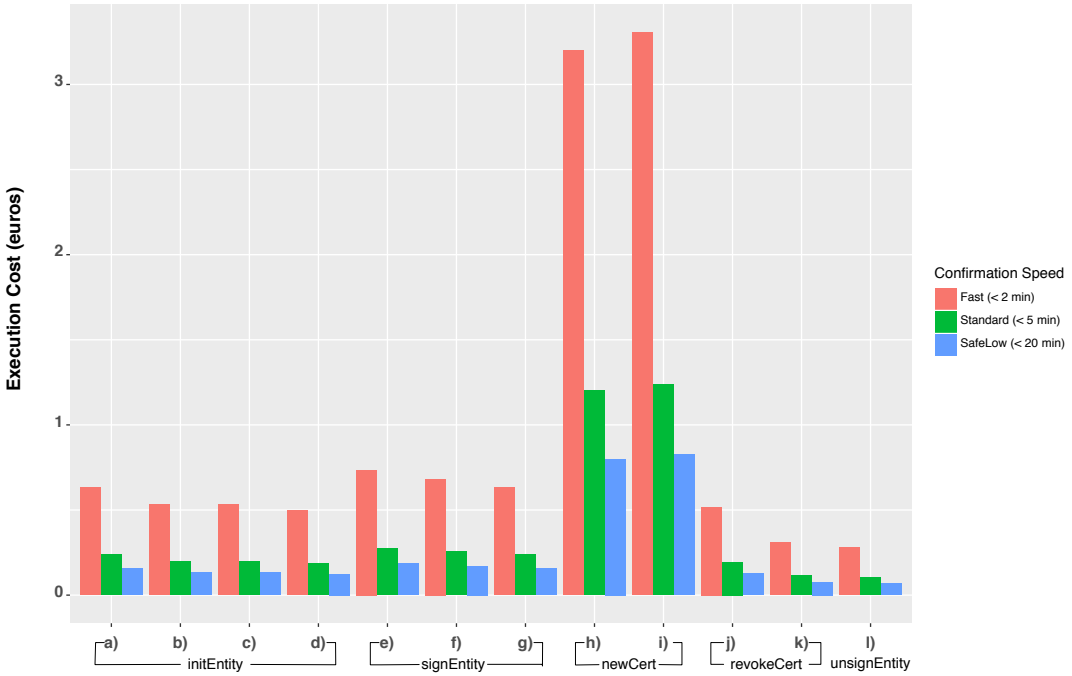



Figure 5.10: Scenario 1 operations costs.

is due to the fact that in the *newCertificate* we are storing much more information in the smart contract state incurring in a bigger gas fee. The *unsignEntity* function in this scenario incurred in a cost of only 0.28 euros, since the unsigned entity (Entity 3) that becomes invalid doesn't have vouched for any entity, meaning that the *checkValidity* function is only called once.

### Scenario 2

The following operations were performed:

a. Entity 0 vouches for Entity 3 (signEntity function);

b. Entity 1 vouches for Entity 3;

c. Entity 2 vouches for Entity 3;

d. Entity 0 vouches for Entity 4;

e. Entity 2 vouches for Entity 4;

f. Entity 3 vouches for Entity 4;

g. Entity 3 vouches for Entity 5;

h. Entity 0 unvouches Entity 3 (unsignEntity function).

In this scenario we wanted to analyze the cost of performing the *checkValidity* function along the descendants of the invalidated entity. We suppressed the entity creation operations from the list of operations being analised.

In Figure 5.12 is depicted the cost analysis of operation exection, and in Figure 5.11 the graph of the web of trust formed.

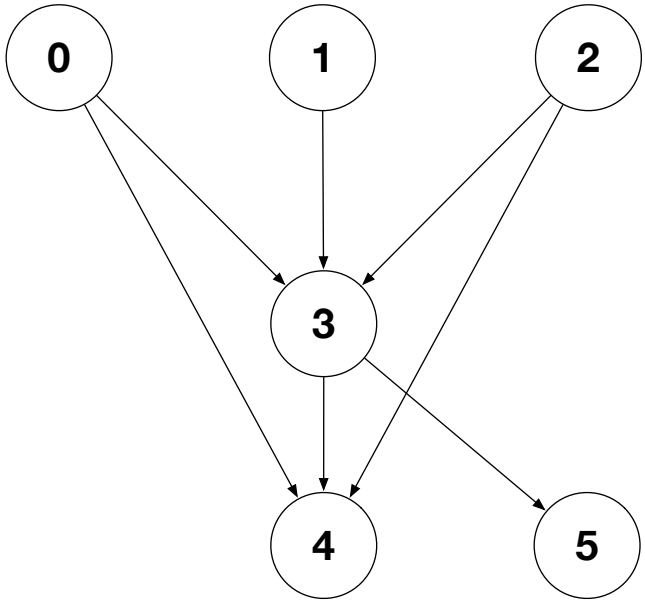



Figure 5.11: Scenario 2 web-of-trust graph.

In this scenario the cost of the execution of *unsignEntity* function – using the ”Fast” transaction confirmation speed as reference – incurred in a cost of 0.35 euros. The cost is slightly higher that in the previous scenario since we are performing an extra two executions of the *checkValidity* function to verify the *Entities 4 and 5*.

#### Overall results

The overall costs of operations execution allows to conclude that is doable cost-wise to build a web of trust system on top of the Ethereum blockchain. The higher execution in all of the operations was in *newCertificate* function which needs to store a higher ammount of data in the smart contract storage. But in that is case is possible to use the Standard or SafeLow confirmation speeds and reduce the execution costs, since generally is not a time urgent operation – as could be a certificate revocation operation. The cost of execute recursively the *checkValidity* in the context of the *unsignEntity* and *signEntity* functions seems to incur in slightly low increments of cost. The cost difference of executing the *unsignEntity* function in Scenario A (calls *checkValidity* once) and Scenario B (calls *checkValidity* three times) is of 0.07 euros. But need to take into consideration that our analized scenarios are relatively simple, if have dozens of nodes with several vouches between each other the cost of executing a simple sign/unsign of a entity could have a considerably higher cost.

The cost analysis in a fiat currency like euro also has the problem of we are analizingthe cost in the

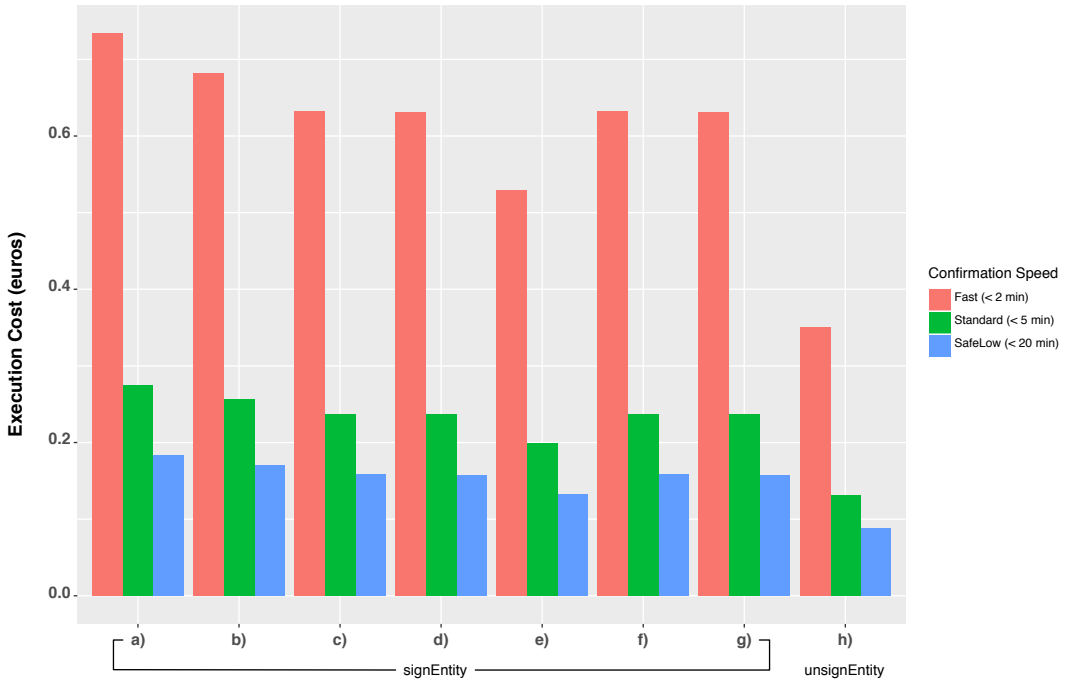



Figure 5.12: Scenario 2 operations costs.

context of the main public Ethereum network. If this certificate is deployed in a private Ethereum network with fewer nodes, and all nodes are mining and receiving ether for every successful block mined, the costs dinamics could be different since there isn't *real-world value* inherent to the ether in this network.

# Chapter 6

# Conclusions

This document describes IDChain, a novel approach to close participation and securing node's communication in DHTs by leveraging Ethereum blockchain's smart contracts. We aimed at providing communication security and DHT access control, by creating an alternative to the classic PKI infrastructure with Certificate Authority entities, with special focus on federated models, as is the case of the reTHINK project. This chapter reflects on these contributions and discusses future work.

## 6.1 Summary

Our approach to improve DHT communications security and access control began with an evaluation of the DHT implementations — with special focus in the Kademlia architecture — and possible attacks, like Sybil and Eclipse attacks. Due to a lack of full proof mechanism to prevent these sort of attacks, we decided to analyze the idea presented by Douceur[12], which states that trusted certification is the only approach that has the potential to eliminate Sybil attacks. We then proceeded to analyze approaches to trusted certification, like Certificate Authorities and Web-of-Trust models.

From our research the two solutions were feasible to apply to a DHT system, but still require some centralized points to store or issue the certificates. We pretended to build an approach that allowed us to minimize trust, i.e to avoid centralized points of control and trust, and started to analyze blockchain technologies. With all the research performed we decided to implement a DHT with a DPKI system using a Web of Trust model, on top of the Ethereum blockchain.

This solution allowed us to provide a decentralized trusted mechanism, which permitted us to minimize trust between DHT participants, prevent Sybil attacks and deal with compromised nodes. Since the main usage of this solution was to secure and close the Global Registry DHT of the reTHINK project, we also decided to build an additional mechanism using the traditional CA infrastructure. This way, it was not only possible to guarantee a fallback mechanism to our IDChain mechanism proof-of-concept, but it was also possible to perform a comparison between the two mechanisms.

We implemented the two solutions using TLS connections between the nodes, each one using different mechanisms to perform certificate validation. In the case of the CA-based architecture, we used

a two-tier CA architecture that needed to be setup in advance to issue nodes certificates. The IDChain mechanism used a smart contract that encoded the necessary functions and rules to recreate a PKI infrastructure, as for example, certificate revocation, certificate fingerprint association and Web of Trust management. In order to improve the IDChain system integration and management, we also built a RESTful API and a management web application.

We performed the two solutions evaluation by deploying 10 DHT nodes to DigitalOcean VMs, spread across several datacenters worldwide. We were able to conclude that performance-wise the IDChain mechanism is a viable option to the CA-based solution. Security-wise the IDChain mechanism could have some weak spots in terms of the smart contract code, taking into consideration that it is designed as a prototype system.

Therefore we have achieved the goal we set out at the beginning of this dissertation: close participation and provide secure node's communication in DHT systems.

## 6.2 Future Work

Many improvements could be done to improve the presented research. In terms of the node's communication in the DHT, one of many improvements it is to try to fix the inability of disable the verification of client's self-signed certificates with TLS mutual-authentication in Node.js. If this could be fixed, the next step would be try to integrate the certificates' verifications we implemented directly into the TLS handshake protocol.

The IDChain smart contract could also have several improvements. The present smart contract can have potential software bugs and edge cases that need to be solved. We need to take into consideration that has the smart contracts deployed to the Ethereum blockchain are permanent, an extended security audit of the smart contract could be performed in order to detect several potential issues. The whole Ethereum developer ecosystem is still in its infancy and better development tools need to be created to improve smart contracts' security. A static typed language and formal verification methods come to mind as tools that could greatly improve the smart contracts security.

New methods of controlling the number of certificates that a single entity can have could also be investigated. For instance, try a dynamic threshold on the maximum number of certificates, that changes accordingly to the total number of certificates in the system, and guarantees an equal distribution of nodes through all the entities participating in the DHT system – for example, each entity could only have 5% of all certificates registered in the system.

A redistribution of funds stored in the smart contract could also be created. Since is necessary to pay for each certificate by sending a monetary transaction to the smart contract address, the smart contract could have a mechanism that could be trigger automatically every $n$ blocks, which redistributes the stored funds equally by all the entities.

Finally, a cost evaluation of the IDChain smart contract with a bigger web-of-trust – around 50 or 100 entities – could be performed in the main Ethereum network.

# 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] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. **41**(6) (October 2007) 205–220

[3] Lakshman, A., Malik, P.: Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2) (April 2010) 35–40

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

[5] 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

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

[7] 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

[8] 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

[9] 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

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

[11] 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)

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

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

[14] 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)

[15] 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

[16] 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)

[17] 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

[18] 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

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

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

[21] 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

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

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

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

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

[26] 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.

[27] 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.

[28] 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)

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

[30] 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

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

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

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

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

[35] Salowey, J., Zhou, H., Eronen, P., Tschofenig, H.: Transport layer security (tls) session resumption without server-side state. RFC 5077, RFC Editor (January 2008) `http://www.rfc-editor.org/rfc/rfc5077.txt`.

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

[37] Cachin, C.: Architecture of the hyperledger blockchain fabric. (2016)

[38] Hearn, M.: Corda: A distributed ledger. Corda: A distributed ledger paper (nov 2016)

[39] Linn, J.: Privacy enhancement for internet electronic mail: Part i: Message encryption and authentication procedures. RFC 1421, RFC Editor (February 1993)