Secure DHT with Blockchain technology

Nuno Filipe Mateus Nogueira

Instituto Superior Técnico, Universidade de Lisboa
nunofilipenn@tecnico.ulisboa.pt

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 closed 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, while also reducing the required 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 authenticated connection establishment between nodes through Transport Layer Security (TLS). We also built a RESTful API and a web application to ease the integration and management of the system.

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

I. INTRODUCTION

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

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

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

On the other hand, we have the worldwide Telco ecosystem that provides an 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 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**, 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 Distributed Hash Table (DHT) technology that stores user identifiers in hyperty services, indexed by a Global User Identifier (GUID).

---

\(^1\)https://rethink-project.eu/
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.

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.

The reTHINK framework is also inherently a federated solution, consequently 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.

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 the mechanisms to perform access-control to DHT nodes and provide secure communications between nodes. The second solution 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 permits 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. This two solutions will have as basis a DHT which 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.

II. BACKGROUND AND RELATED WORK

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). [2]

DHT systems are P2P systems that 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.

Our research will focus on the Kademlia [3] DHT which maps nodes into a balanced binary tree recurring to the XOR operation as a distance metric to perform parallel lookups. Kademlia uses a 160-bit uni-dimensional address space is for nodes’ and keys’ 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.

DHT attacks could be categorized into four categories: a) routing attacks [4], b) storage and retrieval attacks, c) Sybil attack and Eclipse attack.

The Sybil Attack is an attack that exploits a distributed system when it fails to guarantee that distinct identities refer to distinct entities [5]. 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.

There is a number of different defense mechanisms to mitigate this attack: a) centralized certification, b) network characteristics [6][7], c) computational puzzles [8], d) social networks [9][10] and game theory Margolin2008.

Douceur [5] argues that the only way to eliminate the Sybil attack is by having centralized certification.

A Public Key Infrastructure[11][12] 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.

One of the fundamental elements of the Public Key Infrastructure (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. The solutions that implement a PKI infrastructure mainly consist of Certificate Authority (CA) and Web-of-Trust.

The blockchain is a distributed ledger and one of the key mechanisms behind the Bitcoin[13] cryptocurrency. It allows a set of nodes to achieve consensus about the state of a dataset by leveraging a mechanism of proof of work.

Ethereum$^2$ is a protocol for building distributed applications on top of a blockchain, with a Turing-complete programming language, using the smart contract$^3$[14] concept. It uses an

1https://github.com/ethereum/wiki/wiki/White-Paper
internal cryptocurrency, ether (similar to bitcoin) and is used to reward the computational resources used to process the smart contracts and securing the network. 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.

III. ARCHITECTURE

Our main goal is solving 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 was tackled by proposing two different solutions, a CA-based mechanism and a decentralized mechanism which we named IDChain.

The IDChain system should provide the following 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;
- Enable to easily add participants to the DHT;
- Deal with key compromise, by allowing to revoke nodes certificates.

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.

This 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.

A. Vanilla DHT

The vanilla implementation DHT represents the simpler architectural model of the presented solutions. The architecture of this solution is presented in Figure 1.

The DHT nodes will implement a simple Application Programming Interface (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 the DHT system. The DHT node can be used and deploy as part of applications, since it will be required as library by the application code, which then call directly the DHT put/get functions. The DHT nodes will communicate through a insecure channel using TCP or UDP, which are the most common protocols used public facing DHT systems, for example, 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 secure communication protocol between the DHT nodes.

B. 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 doesn’t provide any mechanism to close the participation in the DHT (assuming the DHT nodes are Internet facing servers).

The classic approach to provide a secure communication channel between nodes, is by using the Transport Layer Security (TLS) protocol coupled with a X509 PKI infrastructure, which is summarized in Figure 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 is possible to establish a CA which will issue the certificates of all the SP nodes.

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

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

The node bootstrap process is the same as the vanilla solution, but when establishing the TLS connection between the nodes, 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 this verifications are done when establishing the TLS connections. A possible solution to mitigate this performance decrease is to use TLS Session Resumption[15] mechanism that allow a TLS connection server to resume sessions, avoiding at the same time keeping session state per-client.

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 this verifications are done when establishing the TLS connections. A possible solution to mitigate this performance decrease is to use TLS Session Resumption[15] mechanism that allow a TLS connection server to resume sessions, avoiding at the same time keeping session state per-client.

C. DHT with IDChain mechanism

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

a) DHT based on Kademlia and b) DPKI built on top of the blockchain

In Figure 3 an high-level overview of the architecture is shown. In Section III-C1 and III-C2 an in-depth description of each individual component is given.

The DPKI can be subdivided in three main components: a) blockchain, b) IDChain API and c) IDChain application.

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

1) Distributed Hash Table: Each node in the DHT will have a self-signed certificate that is needed to ensure a secure routing message exchange between nodes in the overlay network.

When a node is routing a message through the overlay network, the peer connectivity is done using TLS connections with mutual authentication, so that the two peer certificates can be exchanged and verified.

In order to verify the authenticity of the certificate, the peer identifier of the message sender should be equal to the peer identifier in the sender certificate. But this verification is not sufficient to guarantee the validity of the certificate, since a peer this way can impersonate several identities.

So was necessary to implement a mechanism similar to certificate pinning. Therefore the peer should also check if there is a correspondence between the certificate fingerprint and the peer identifier registered in the blockchain.

2) 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 have 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.

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

The DPKI will have the following functionalities: a) register and store certificates associating a node identifier with its certificate fingerprint, b) revoke compromised certificates and c) allow users to query the blockchain, in order to retrieve these associations.

Since we wanted to build this DPKI mechanism on top of TLS, complementing it, we used some mechanisms that are used in Certificate Pinning4.

3) IDChain Smart Contract: The logic and set of rules that compose the DPKI are stored in a blockchain smart contract.

4https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning
The smart contract that we created has the main objective of associate an peer identifier to a valid certificate, in a way that any system can easily query for and verify that association. Since we were trying to build this DPKI under a federated model, is was still necessary to define the trust mechanism in this system. So we built the trust mechanism relying on a web of trust model built using the smart contract. The minimum functionalities that the smart contract should provide are:

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

As is shown in Figure 4, a newly created entity to be accepted should be vouched by a minimum number of entities. One peculiar aspect of this architecture, consequence of the federated model, is that each entity in the system is able to create several node certificates. This can undermine smaller entities participating in the system, and could even allow one single entity to control the whole DHT, by generating unlimited node certificates.

The IDChain API is backed up by a relational database (using Structured Query Language (SQL)) where all the transactions related with the system will be stored. This kind of associations and structure is harder to obtain when using the blockchain client directly, since the client only deals with transactions at a lower level, without attaching a meaning at a smart contract basis to each transaction. In order to store the data in the database, blockchain events are used. Is possible to trigger events in smart contracts that, when are executed in a transaction context, will notify the blockchain client, i.e. when a client attaches a valid block – with transactions related to this specific smart contract – to the blockchain the client will be notified of this event. This allow to mirror the transaction information related with our smart contract to the relational database.

5) **IDChain Management Application**: We also built an application that allows to do all the operations related with the IDChain system. It is a web application, that interacts directly with the IDChain API.

The main functionalities revolve around entity and certificate management: view entities vouchers, create new node certificates, view the transactions related with each entity, etc.

The main inspiration for this system are the web-based blockchain explorers⁵ that exist for blockchains like Ethereum

---

⁵https://blockexplorer.com/
The web application was built using an Single Page Application (SPA) architecture, leveraging browser Javascript frameworks.

IV. IMPLEMENTATION

A. Vanilla system

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.

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;

B. CA-based system

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

C. IDChain system

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.

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.

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.

2) **Smart contract:** One mechanism built-in in the Ethereum blockchain 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 by the smart contract’s functions.

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

**Algorithm 2 Contract global state initialization.**

```plaintext
counterId ← name  // certificate identifier counter
revocations ← emptylist  // identifier of revoked certificates
certificates ← emptymap  // map associating a peer identifier to a certificate
entities ← emptymap  // map associating a Ethereum account address to a entity
```

In our implementation there is a one-to-one relationship between 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` function 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 bootloader 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.

After an entity is created it is now possible to create certificates that will be associated with this entity. The `newCertificate` function is called when an entity wants to associate a new certificate. 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.

The certificate revocation functionality is implemented by the `revokeCertificate` function. 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 is to be revoked.

The main aspect of the IDChain smart contract is the possibility to build the trust between the different entities. The functions `signEntity` and `unsigEntity` 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 `unsigEntity` 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 at the end of their execution.

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

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.

4) **Application**: The data is fetched from the IDChain API, using the web browser `fetch API`.

We implemented the following functionalities: 
- `a)` Register new certificate (with a certificate drag-and-drop feature). 
- `b)` sign/unsign new entities, 
- `c)` list certificates registered by the entity, 
- `d)` view all transactions associated with the entity.

V. **Evaluation**

In order to evaluate the implemented solution, several tests were done to assess protocol correctness and measure system performance.

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.

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. We ran 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.

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

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 ran on a server – located in Lisbon – 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 to the Digital Ocean nodes.

1) **Load tests**: The Figures 5- 8 represent the write and read load tests evaluation results.

The graph from Figure 5 (write requests) and Figure 6 (read requests) represents the effective request rate in function of the demanded request rate, for each of the implemented solutions. Each point in the graphs, illustrates the average request rate of the 5 different test executions performed, for the specific demanded request rate.

In the Figures 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 $(0.1 \text{ req/s})$ for 5 repetitions performed for this test, demanding a request rate of $0.1 \text{ req/s}$.

In Figures 5 and 6 is possible to verify that in the $[0.1, 10]$ request/s demanded request rate range, all the 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 keep up more closely with the demanded request rate. The vanilla DHT implementation is able to perform $\approx 90 \text{ requests/second}$, whereas the CA-based and IDChain solutions become saturated at $\approx 17 \text{ requests/second}$, in the case of write requests, and at $\approx 20 \text{ requests/second}$ in the read requests. 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. 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.

6https://www.digitalocean.com/
The graphs from Figures 7 (write requests) and 8 (read requests) represent the average request response time in function of the effective request rate, for each of the implemented solutions. Each point in the graph, illustrates 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, ≈ 2 times faster response times.

The average response time of the CA and IDChain based DHT implementations remain approximately constant, only increasing when an ≈ 15-17 requests/second effective request rate is achieved in write requests test, and ≈ 15-20 requests/second effective in the case of read requests test. These correspond to a demanded request rate in the [20, 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, in the case of write requests is achieved. In the case of the read requests, an average response rate of 3302.602 ms, when a request rate of ≈ 91 requests/second is achieved. Those correspond to a demanded request rate of 100 request/second. 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.

CONCLUSION

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.

We analyzed 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.

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 an 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.

VI. 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 is to try to fix the inabilty 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. An extended security audit of the smart contract could be performed in order to detect several potential issues. 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.

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.

REFERENCES