Laboratory For Data Security (LDS)

MedChain: A Distributed Authorization and Authentication System for Medical Queries

Master Semester Project Report

Author:
Zahra Farsijani
School of Engineering, Electrical Engineering

Professor:
Jean-Pierre Hubaux
Supervisors:
Juan Troncoso-Pastoriza
David Froelicher
Mickaël Misbach

Monday 27th January, 2020

Abstract

There is great body of research that depends on use and analysis of clinical and health data as well as sharing them among multiple institutions in some way. This gives rise to some concerns and issues regarding the privacy of data owners due to researchers' access to such sensitive data. A number of software solutions have been developed in order to address this problem and allow privacy-preserving analysis and sharing of sensitive medical data among many institutions such as MedCo [1], being the very first operational one. However, when it comes to authentication and authorization of data users, MedCo exhibits some limitations. First, in MedCo, authentication and authorization are central which introduces a single point of failure within the network. Second, each participant in MedCo cluster, runs its own access management process.

In this project, we propose "Medchain: a distributed authorization and authentication system for medical queries" that overcomes the drawbacks present in MedCo. Medchain works seamlessly with MedCo and complements its full workflow by handling the access management. Furthermore, Medchain offers auditability as it records all the medical queries from the users in an immutable ledger, i.e., blockchain.

Contents

1 Introduction

1.1 Problem Definition and Objectives

In this project, we aim to design and implement a distributed authentication and authorization system for medical queries, called Medchain. We also intend to make this system specifically compatible with the ecosystem of MedCo. MedCo is an existing technology that enables privacy-preserving analysis and sharing of medical data. MedCo already has authentication and authorization mechanisms in place, however, they have some drawbacks which Medchain tries to address. First, in the current implementation of MedCo, both authentication and authorization are centralized. This introduces a single point of failure in the whole ecosystem and has to be avoided. Second, hospitals and medical institutions that participate in MedCo cluster have their own identity providers and run access management protocols individually as these processes can not be done in a federated manner. Our proposed software system, Medchain, offers solutions for such shortcomings. It also puts forward other sought-after properties, such as auditability by recording all the queries submitted to it in a blockchain, which is explained in more details later in this report.

1.2 Report Outline

In this report, we first provide some background information on the underlying technologies used in Medchain deployment in Section 2. Next, in Section 3, design and architecture of Medchain is presented and discussed. In Section 4, we give a full descriptions of our solution, its software design and building blocks. Later, in Section 5, we discuss the performance of our software solution and its limitations. In Section 6, we discuss the future opportunities to improve the solution and, finally, we finish with conclusions in Section 7.

2 Background

In this section, we provide some background information about the building blocks of Medchain and some technologies used in its development.

2.1 Cothority

The goal of this project is to implement a system that offers services in a distributed manner. To achieve this, we used collective authority in Medchain. Collective authority (cothority) available at [2] is a project developed and maintained by DEDIS lab at EPFL. Its objective is to provide various frameworks for development, analysis, and deployment of decentralized and distributed (cryptographic) protocols. In order to be able to access the services provided by cothority, one should run a set of servers that run cothority protocols. Such set of servers is referred to as a **collective authority** or **cothority**. Furthermore, every single server in cothority is called a cothority node or a **conode**. The code in cothority repository allows the developers to access the services of a cothority and/or run their own conodes.

We decided to use conodes due to the distributed protocols they offer and thus every Medchain server is a conode. In the following sections, we look at some of the most important building blocks of Cothority as well its services and protocols that play functional role in Medchain.

2.1.1 Conode

As it was mentioned earlier, a **conode** is a server and conodes are linked together to form a cothority. Clients need to run conodes, either locally for local tests or in public mode in order to access decentralized protocols and services offered by cothority.

To operate a conode, one needs to correctly set up a host and run the conode program. The reader is referred to conode documentation and instructions on how to setup and use it found at [3].

2.2 ByzCoin

ByzCoin in one of the services offered by cothority and there are many cothority applications that use ByzCoin as there building block. ByzCoin is a scalable Byzantine fault tolerant (BFT) consensus algorithm for open decentralized blockchain systems. It uses the skipchain [4] as its underlying data-structure for storing blocks and thus has a distributed ledger holding keys and values. It also implements pre-compiled smart contracts. The **Omniledger** paper [5] describes the protocol that ByzCoin implements. The reader is referred to [6] for further details about ByzCoin: how to use it, its building blocks, etc.

There are various basic data structures used in ByzCoin. Below, we provide a brief overview of the most relevant ones:

- **Instructions**: An Instruction is created by a client and it will be executed in ByzCoin. It can be a `Spawn`, `Invoke`, or `Delete` command which results in a `StateChange` if it is accepted by ByzCoin. The purpose of each command is described below:
  - `Spawn`: create a new instance
  - `Invoke`: call a method of an instance such as `update`
  - `Delete`: remove an instance
- **ClientTransaction**: A *ClientTransaction* holds one or more instructions and is sent by a client to one or more conodes in the roster.
- **StateChange**: Instructions (i.e., ClientTransactions) sent by a client correspond to a contract/object. The execution of these instructions (i.e., the calls to the contracts/objects) results in 0 or more changes in the global state referred to as **StateChange**.
- **Distributed Access Right Controls (Darcs)**: Darcs offer means to control access rights to the available resources in ByzCoin. Since Darc is one of the most important building blocks of Medchain, we have dedicated a separate section to it. Please refer to Section 2.3.3 to learn more about the Darcs.

2.3 Smart Contracts and Instances

A smart contract is a piece of code that implements a protocol which guarantees the automatic execution of a contract when the specified conditions are met. In Byzcoin, contracts are pre-compiled and the conode holds the binary of the smart contract. In general, we can say that the smart contract is a collection of methods that provides the client with the API to interact with the skipchain (i.e., the ledger that holds the instances of the contract) through transactions. In

other words, contract interprets the methods, namely the instructions, sent by the client for the cothority server (conode). The execution of a client instructions by smart contract results in a change in the shared **global state**, called a `StateChange`. In order to commit the transactions onto the ledger, the threshold of agreeing conodes must be reached. A conode can hold various contracts at the same time. Each contract is identified by a string pointing to it called the "contractID".

Byzcoin uses coins as the mining reward. At the input a list of all available coins it provided to the contract. After the contract is run, it needs to return the new list of coins that are available.

Below, you can find the list of input arguments given to a contract and its output arguments:

Input arguments:

- pointer to database for read-access
- Instructions sent from the client
- key/value pairs of available coins

Output arguments:

- one StateChange (may be empty)
- updated key/value pairs of remaining coins
- error that will abort the clientTransaction if it is non-zero. If any contract returns non-zero error, the state will not be changed.

2.3.1 Contract Instance Structure

In ByzCoin, the global state holds the instances of contracts and is split by the Darcs that define the access to control the corresponding instances. Every instruction sent to an instance must resolve the rule in the governing Darc. Every instance is stored with the following information in the global state:

- **InstanceID** is a globally unique identifier of the instance.
- **Version** is the version number of this update to the instance, starting from 0.
- **ContractID** points to the contract that will be called if the instance receives an instruction from the client
- **Data** is interpreted by the contract and can change over time
- **DarcID** of the Darc that controls access to the instance.

2.3.2 Interaction between Instructions and Instances

Once a client sends an instruction, he/she indicates the InstanceID to which it corresponds per an instruction argument. ByzCoin tries to first authorize the instruction through the process described in Section 2.3.3, it then uses the `InstanceID` to look up the responsible contract for the instance and then send the instruction to that contract.

2.3.3 Distributed Access Right Controls (Darcs)

Darcs enable us to control access to contracts/objects in ByzCoin and thus can be used for authorization and/or authentication in software systems. Instead of using a password or a public key for authentication, Darcs allow us to implement access control rules that can be *evolved* based on a threshold number of keys. This essentially means that instead of having a static set of rules (e.g., a fixed list of identities that are allowed to access a resource), we can evolve the existing rules any time and thus achieve dynamism in our access right definitions.

A Darc is a data structure that has a set of rules that define what permissions are granted to any identity (i.e., public key) as pairs of action/expression. An example of a Darc action/expression pair is: `invoke:ContractX.update - "ed25519:<User1's public key> | "ed25519:<user2's public key>""`, which means that either of users 1 or 2 can update the instances of `ContractX`.

Every Darc is stored with an `InstanceID` that is equal to the Darc's `baseID`. Once a Darc is evolved (i.e., its rules are updated, see later), this `InstanceID` is overwritten with the new value. Whenever the client sends an instruction for execution, he/she has to also provide the `InstanceID` of the Darc (i.e., DarcID) that governs the corresponding contract instance, e.g., `ContractX`, that is to be affected by the instruction. Thus, the Darc comes into play and checks if the client has the right to execute the instruction and consequently, approves or rejects the instruction on the instance. This scenario can be summarized into the steps below:

1. Client sends the following instruction to ByzCoin (some fields are omitted for clarity):

```
InstanceID: <Darc's InstanceID>,
    Invoke:{
        ContractID: ContractX,
        Command:    "update",
        Args:{
                Name:  query.ID,
                Value: []byte(query.Status),
            },
    }
```

2. ByzCoin finds the Darc instance using the Darc's `InstanceID` provided by the client in the instruction.
3. ByzCoin creates an `Invoke:update` instruction on ContractX's instance to update the key-value pair provided in `Args`.
4. ByzCoin checks the Darc found in step 2 and verifies that the request corresponds to the `invoke:update` rule (i.e., expression) in the Darc instance for the given identity (the client who creates this transaction).

A Darc can be updated by an *evolution* process: the identities that have the evolve permission (i.e., `darc:evolve`) in the Darc create a signature and sign off the new Darc. There is no limit to the number of evolutions that can be performed and evolutions result in a chain of Darcs, also known as a path. A path can be verified by starting at the oldest Darc (also known as the base Darc), walking down the path and verifying the signature at every step. Delegation is

also possible in Darcs. That is, a Darc can be given `evolve` permission to evolve other Darcs and thus evolve the path.

2.4 MedCo

MedCo [1] is the first operational system that enables the exploration and analysis of distributed (sensitive) medical data in a privacy-preserving manner. By using strong collective encryption provided by collective authority [7], MedCo is able to offer data privacy. MedCo aims to distribute trust among various medical data providers (such as medical institutions) so that their data can be used and queried by external users while privacy is preserved.

MedCo is built on top of existing and open-source technologies: (i) i2b2 [8] is a clinical research platforms and together with SHRINE [9] they enable clinical data exploration; (ii) UnLynx [10] allows distributed and secure data processing in MedCo.

3 Design and System Architecture

3.1 General Overview

In this section, we will look at the architecture of Medchain system. We first begin by looking at the requirements that our software solution should be able to fulfil. Then, we discuss how our solution meets its requirements and objectives.

3.2 Requirements

Medchain aims to offer distributed identity and access management mechanisms for (medical) data sharing systems. However, the main goal of this project is to apply Medchain for a specific medical data system called MedCo described in Section 2.4. To achieve this, Medchain has to fulfil the following tasks and requirements:

1. Offer its services in a distributed manner
2. Offer access management and user authentication
3. Authorize users and queries
4. Enable auditability by recording a full history of queries in blockchain
5. Work seamlessly with MedCo

In the following sections, we study how each of the above-mentioned requirements are fulfilled in Medchain.

3.2.1 Distributed Services in Medchain

As it was mentioned in the previous section, the first goal of Medchain is to offer services in a distributed manner. To achieve this, we used conodes. Conodes make the foundation for Medchain. They support distributed protocols and services provided by the onet library (a part of Cothority project developed by DEDIS at EPFL). Consequently, Medchain is able to offer access management in a distributed manner.

3.2.2 Authentication in Medchain

The goal of authentication is to verify the identity of the user who wants to access a specific resource such as data. In Medchain, user authentication is delegated to Keycloak [11] which is the identity provider. This means that once Medchain receives a query from a user, we can simply assume that he/she is already authenticated and his/her identity and the user identity is submitted to Medchain as part of the query. We will discuss this in more details in Section 3.4, where the flow of query in Medchain is described comprehensively.

3.2.3 Authorization in Medchain

The goal of authorization is to control the user access to resources (e.g., data) based on his/her access level. In Medchain, we use Darcs (described in Section 2.3.3) to enable user authorization and access management.

3.2.4 Auditability of Medchain

Auditability in Medchain means that there should be a mechanism in place so that the users can track all the queries submitted to Medchain server. Consequently, trace of any data misuse, data abuse, security/privacy breach, etc. can be found and mitigated later.

In Medchain, we use blockchains to offer auditability. The blockchain framework we used in Medchain is ByzCoin (see Section 2.2). Every query sent by the user and its status, whether it is authorized or rejected, is recorded in the immutable ledger.

3.3 Medchain High-level View

Figure 1 shows the full architecture of a Medchain node. Every Medchain node is a conode and it contains three important building blocks. Below you can find more about each of these building blocks and their purpose:

- **Smart contract**: offers user API and distributed access management
- **Darcs**: manage user's access to resources
- **Blockchain**: auditability

Additionally, every Medchain node has 1 or more projects in it. A project is an abstract notion that wraps around a specific data resource and restricts access to it. Implementation-wise, every project is, in fact, a Darc that controls the access rights concerning a specific database.

3.4 Flow of Queries in Medchain

Figure 2 illustrates the full architecture of Medchain and its detailed workflow starting from query creation and submission by the client until the query is executed.

In this figure, there are three hospitals each running their own local Medchain server (conode). Now, focusing only on the user at Hospital 1, we describe the enumerated steps in Figure 2 as following:

1. **User Login**: Client receives the token (i.e., its identity) from Keycloak.

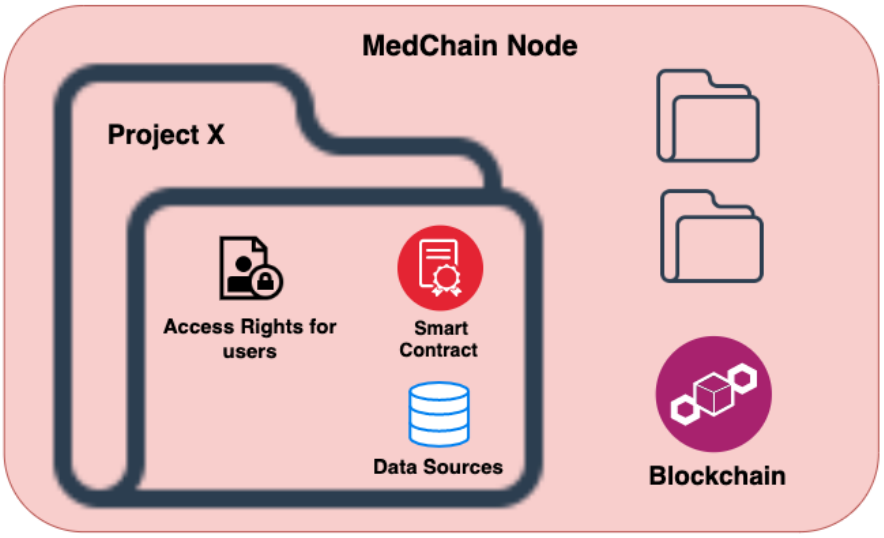

Figure 1: The structure and building blocks of a Medchain node.

2. **Query Creation & Submission**: User spawns a query. The query is submitted to MedCo-connector (i.e., service provider).
3. **User Authentication**: MedCo-connector verifies the token (i.e., the identity) of the user.
4. **Submission of Query To Medchain**: MedCo-connector sends the query that contains the user's identity (i.e., part of the token) to Medchain server. The query that Medchain receives has a structure as below: Query's structure is as below:

   `{ID:<Query ID>,Status:<Query Status>}`

   The query itself has the following format:

   `<user_id>:<databaseX>:<type of query>`

   Where `<user_id>` is part of the user's token provided by Keycloak, `<databaseX>` is the name of the database (i.e., the project) user is querying, and `<type of query>` is what the user is trying to access from the database which can be one of the below:

   - `patient_list`
   - `count_per_site`
   - `count_per_site_obfuscated`
   - `count_per_site_shuffled`
   - `count_per_site_shuffled_obfuscated`
   - `count_global`

- `count_global_obfuscated`

Finally, `Status` is the status of the query in Medchain and it could be: `"Submitted"`, `"Authorized"`, or `"Rejected"`.

5. **Query Transaction Creation**: Medchain creates a transaction to spawn a `Query` (i.e., an instance of `queryContract`) and signs it. The transaction looks like below:

```
CreateTransaction(byzcoin.Instruction{
    InstanceID:    byzcoin.NewInstanceID(<Project X Darc ID>),
    Spawn: &byzcoin.Spawn{
        ContractID: queryContract,
        Args: byzcoin.Arguments{
                {
                    Name:  <user_id>:<databaseX>:<type of query>,
                    Value: []byte("Submitted"),
                },
        },
})
```

6. **Query Authorization**: The original Medchain node broadcasts the transaction proposal to the rest of the Medchain nodes in cothority network so that it is recorded in the ledger. Each Medchain server fetches the project Darc by the provided ID and checks if the query can be executed based on the identity of the user who submitted the query, what the query is, and the target project according to `Project X Darc` and the associated smart contract (i.e., `queryContract`).

   If a number of Medchain servers in the network are down and/or the responsible entities to accept or reject a query are not present (e.g., a manager at the hospital) at the time of transaction broadcast, the transaction execution is deferred until the threshold number of entities can react to it and either reject or accept it.

   If the query is authorized by the Darc, and all Medchain servers approve it, the `Status` of the query is updated to `Authorized` and if it is rejected, the `Status` is set to `Rejected`. In any case, the query and the result of this stage is written to the ledger.

7. **Query Execution**: In the mean time, MedCo-connector is waiting for a reply from Medchain nodes. If the query is authorized, the query is directed to MedCo-connector in order for execution; otherwise, MedCo-connector receives a rejection message from Medchain. (In fact, MedCo-connector, i2b2 server, and MedCo-UnLynx all take part in query execution step, however, because the focus here is on Medchain, we overlook the details related to the query execution and only resort to high-level descriptions.)

8. **Commit Final Query Status To The Ledger**: After MedCo-connector executes the query, it submits the new query status (e.g., "Executed") to Medchain server. Then, Medchain creates a transaction to update the status of the query to `Executed`. Finally, this transaction is added to the ledger.

   Starting from step 5, the query's `InstanceID` is preserved and using this `InstanceID`, we can keep track of the query and update its status accordingly. This is explained in more

details in Section 4.2.3.

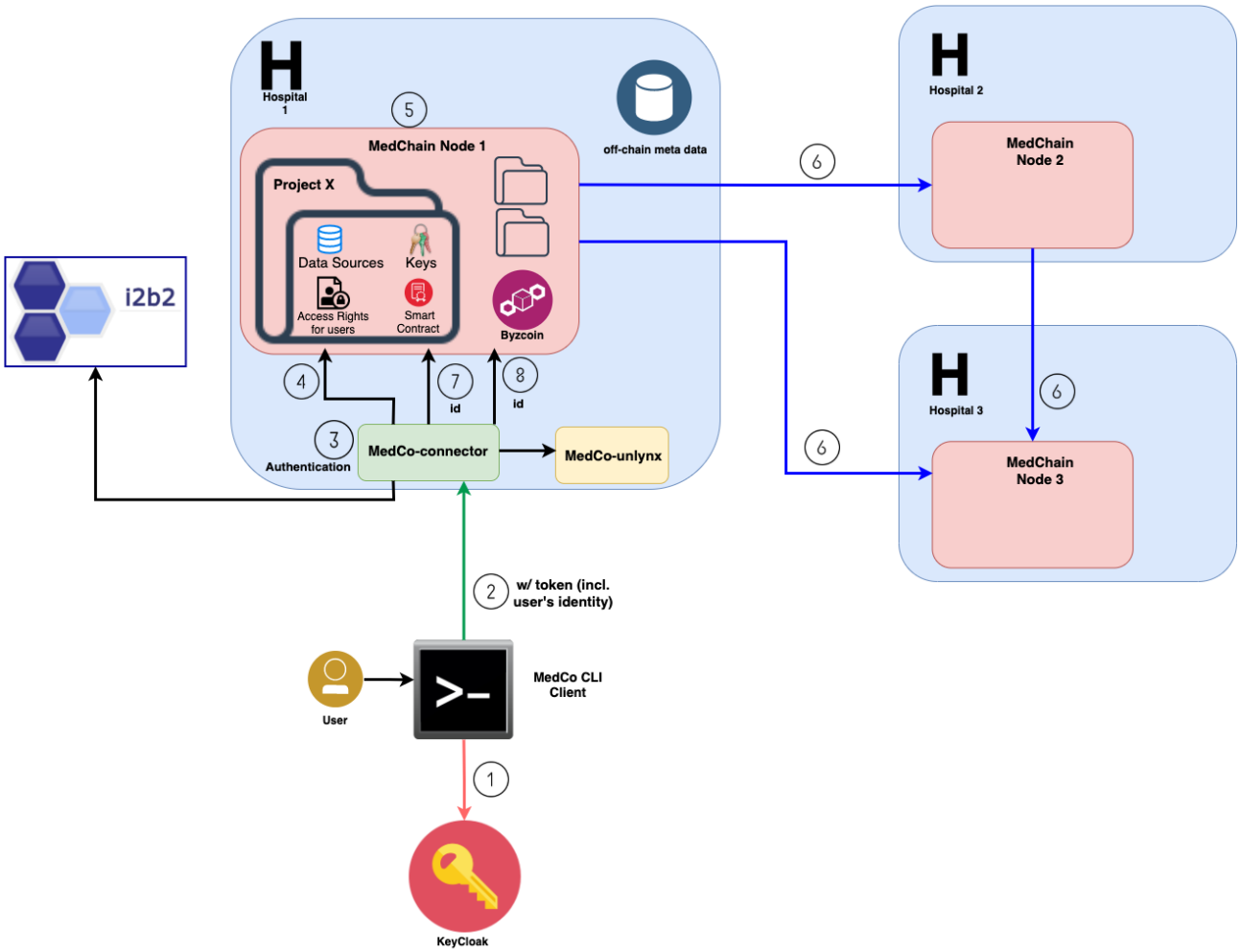

Figure 2: Medchain architecture: workflow of queries in Medchain.

4 System Implementation

The full Medchain code, it documentation as well as instructions on how to run it is available in Medchain Github repository. In this section, we aim to provide details about the code structure of the Medchain and not instructions on how to use the code.

4.1 Conode

Every Medchain node is a cothority server, known as a conode. There are several ways to initialize a conode that are described in conode documentation found at [3]. In this project, we used configuration setup with the command line to setup our network of 3 local conodes in an interactive manner. Also, in order to use the conodes we followed the instructions to run them.

The main point to note here is that once a smart contract is developed and imported into a conode, it is available and can be used by the client. This means that the user does not need to bootstrap the smart contract or install it.

4.2 API

In this section, we provide details about some building blocks of the Medchain Application Programming Interface (API).

4.2.1 Smart Contract

The smart contract provides the user with the API to interact with the skipchain (i.e., the ledger) by adding, updating, or removing the instances of the contract from the ledger.

We developed the smart contract in Go using the template contract provided in cothority_template repository called ByzCoin example which is a simple key-value contract, meaning that every instance of the contract that is recorded in the ledger is a key-value store data structure. Medchain smart contract is identified by its name "**queryContract**". This contract is distinguishable from any other contract used in Medchain deployment such as Darc contracts (see Section 4.2.3). The client uses this ID in order to define which contract instance he/she wants to manipulate. The queries submitted to Medchain are recorded in the ledger as instances of `queryContract`. The structure of these queries is shown below:

```
type Query struct {
    ID     string //<query_id>:<user_id:databaseX>:<type of query>
    Status string // "Submitted", "Authorized", or "Rejected"
}
```

As it is shown above, the key of a queryContract instance is the ID of the query and the value is its status.

To serve the purposes of Medchain, we changed some functionalities of the template contract so that it is adapted to Medchain ecosystem. In ByzCoin, the smart contract has to implement three API methods: `Spawn`,`Invoke`, and `Delete`. The very first instance of a contract is created by creating a transaction using `Spawn` as the instruction. By this transaction, the user sets the query ID as the key of the instance and its status as the value. Later, the user can manipulate this instance of the contract by calling an `Invoke` on it. The `Invoke` method in `queryContract` implements two methods itself: `update` and `verifystatus`. Using update, the user is able to retrieve a specific contract instance (i.e., query) from the skipchain by its key (i.e., query ID) and update its value (i.e., the status of the query) if it already exists in the skipchain, however, if that is not the case, a new instance of the contract will be spawned using the provided key-value pair. `verifystatus` method is used by Medchain server itself to retrieve a query from the ledger and verify its status in a similar manner to `update` method. In Medchain, we decided not to implement the `Delete` function in contract as we do not want the user to be able to remove a query (i.e., an instance of the contract) from the global state.

Whenever a contract instance is created (spawned) it is allocated an `InstanceID` that is based on the ID of the Darc contract governing it (please refer to Section 2.3 for more details). Later, this instance of the contract is retrievable and authorized by the Darc controlling it using this `InstanceID`.

4.2.2 Deferred Transactions

In a real world scenario, it is not always possible for hospitals (running Medchain servers) to approve queries they receive from other Medchain servers instantly, for example, the hospital manager may not be present or the local Medchain server may be down. Thus, the query will be rejected as it will not receive the threshold number of signatures from other Medchain servers in the network it needs so that it is deemed as approved at the time it is created. There should be a mechanism in Medchain to handle this issue. **Deferred transactions** can be used as a solution for this problem. As the name suggests, deferred transactions allow a transaction to remain idle until it receives the threshold number of signatures it needs to be written in the ledger.

In order to enable deferred transactions in ByzCoin server, the developer should define a special method in the smart contract, namely, `VerifyDeferredInstruction`, which is not implemented in a `BasicContract` (i.e., the basic data structure that all contracts implement by default). In other words, types which embed `BasicContract` must override this method if they want to support deferred transactions (using the `Deferred contract`).

To enable deferred execution of a `queryContract` instance, the following steps are taken:

1. User spawns a `queryContract` instance (This is the proposed transaction).
2. User spawns a `deferred_contract` instance with query instance as its arguments.
3. Signers sign the proposed transaction and invoke an `addProof` on it.
4. User invokes an `execProposedTx` on proposed transaction to execute it.

4.2.3 Darcs

In ByzCoin, Darcs are used to enable authorization and access management and are, in fact, smart contracts. The only difference between a Darc and a general smart contract is that a Darc supports `actions` and `expressions` that are used to define a set of rules in a Darc.

ByzCoin offers some Darcs in its `darc` library such as `SecureDarc`, however, the developer can also develop his/her own Darc contract. In Medchain, we use `SecureContract` and have customized it to meet the requirements of Medchain.

`SecureDarc` contract defines access rules for all clients using the Darc data structure. Upon starting a cluster of Medchain servers, a new ByzCoin blockchain and a `genesis Darc` instance are created. The `genesis Darc` indicates what instructions need which signatures to be accepted. Below, you can see how the `genesis Darc` we use in Medchain looks like:

```
- Darc:
    -- Description: "genesis darc"
    -- BaseID: darc:<ID_genesis_darc>
    -- PrevID: darc:<ID_darc>
    -- Version: 0
    -- Rules:
        --- invoke:config.update_config - "ed25519:<ID_client>"
        --- spawn:darc - "ed25519:<ID_client>"
```

```
        --- invoke:darc.evolve - "ed25519:<ID_client>"
        --- invoke:darc.evolve_unrestricted - "ed25519:<ID_client>"
        --- _sign - "ed25519:3<ID_client>"
        --- spawn:naming - "ed25519:<ID_client>"
        --- spawn:queryContract - "ed25519:<ID_client>"
        --- invoke:queryContract.update - "ed25519:<ID_client>"
        --- invoke:queryContract.verifystatus - "ed25519:<ID_client>"
        --- _name:queryContract - "ed25519:<ID_client>"
        --- invoke:config.view_change - "ed25519:<ID_server>
            | ed25519:<ID_server> | ed25519:<ID_server>"
    -- Signatures:
```

In the above `genesis Darc`, we can see the pairs of `action/expression` that define different rules. For example, `spawn:queryContract - "ed25519:<ID_client>"` where `ID_client` refers to the ID of the client who is granted the permission for the action `spawn:queryContract`. Darc expressions are a simple language for defining complex policies. For example, in the rule `invoke:config.view_change - "ed25519:<ID_server> | ed25519:<ID_server>"`, an "`or`" expression has been used among the IDs of Medchain cluster servers.

Now, we create Darcs for different projects (e.g., `Project A` and `Prject B`) in Medchain. We assume that each project has 3 clients (i.e., one client per Medchain server). We create new Darcs and define rules (i.e., action/expression pairs) for them. As an example, the Darc for project A is given below:

```
- Darc:
    -- Description: "Project A Darc"
    -- BaseID: darc:<ID_darcA>
    -- PrevID: darc:<ID_genesis_darc>
    -- Version: 0
    -- Rules:
        --- _evolve - "ed25519:<ID_owner>"
        --- _sign - "ed25519:<ID_client1>" |
        "ed25519:<ID_client2>" | "ed25519:<ID_client3>"
        --- spawn:queryContract - "ed25519:<ID_client1>" |
        "ed25519:<ID_client2>" | "ed25519:<ID_client3>"
        --- invoke:queryContract.update -
        "ed25519:<ID_client1>" | "ed25519:<ID_client2>" |
        "ed25519:<ID_client3>"
        --- invoke:queryContract.patient_list -
        "ed25519:<ID_client1>" | "ed25519:<ID_client2>" |
        "ed25519:<ID_client3>"
        --- invoke:queryContract.count_per_site -
        "ed25519:<ID_client1>" | "ed25519:<ID_client2>"
        | "ed25519:<ID_client3>"
        --- invoke:queryContract.count_per_site_obfuscated -
        "ed25519:<ID_client1>" | "ed25519:<ID_client2>" |
        "ed25519:<ID_client3>"
```

```
        --- invoke:queryContract.count_per_site_shuffled -
        "ed25519:<ID_client1>" | "ed25519:<ID_client2>" |
        "ed25519:<ID_client3>"
        --- invoke:queryContract.count_per_site_shuffled_obfuscated -
        "ed25519:<ID_client1>"
        --- invoke:queryContract.count_global -
        "ed25519:<ID_client1>"
        --- invoke:queryContract.count_global_obfuscated -
        "ed25519:<ID_client1>"
    -- Signatures:
```

In the above Darc for Project A, we have defined rules using different types of actions and expressions. For example, all users can take action `invoke:queryContract.patient_list` and in order for it to be approved, the signature of **any** of the signers is enough as an `OR` expression is used among signers. However, only client 1 is given permission to query `count_global` from database A (i.e., the action `invoke:queryContract.count_global`).

Now, let's see how the Darc for Project A can authorize an action. The first step is that the client sends a spawn instruction to Project A Darc contract. Then, the client asks the instance to create a new instance with the `contractID` of Medchain smart contract, i.e., `queryContract`, which is different from the ID of the Darc instance itself. The client must be able to authenticate against a `spawn:queryContract` rule defined in the Project A Darc instance which is indeed the case according to Project A Darc definition above. The transaction that client creates looks like below:

```
instr := byzcoin.Instruction{
    InstanceID: byzcoin.NewInstanceID(<Project A Darc ID>),
    Spawn: &byzcoin.Spawn{
    ContractID: "queryContract",
    Args: byzcoin.Arguments{
    {
        Name:  < query ID including the action>,
        Value: []byte("Submitted"),
    },
        }
    },
    SignerCounter: c.nextCtrs(),
}
```

The new instance spawned will have an instance ID equal to the hash of the Spawn instruction. The client can remember this instance ID in order to invoke methods on it later. In Medchain, we also use `contract_name` of ByzCoin. This contract is a singleton contract that is always created in the genesis block. One can only invoke the naming contract to relate a Darc ID and name tuple to another instance ID. Using this contract, the client does not need to store instance IDs as long as they are named and thus, makes it easier for the client to use the instance IDs of the contracts.

After this step, a `queryContract` instance spawned by the user (which is in fact the query submitted to Medchain from MedCo) is bound to Project A Darc and is governed by it; thus,

the Darc can check for the authorizations of the action the client is trying to take.

4.2.4 Client Implementation

To implement a client that can interact with Medchain, we used the default clients implemented in ByzCoin Client library. The `Client` is a structure that communicates with the ByzCoin service and interacts with it.

4.2.5 Services

In Cothority, a Service is a long term entity that is created when a conode is created. It serves different purposes:

- serve the client requests
- create and launch protocols in the Overlay network
- broadcast to and receive information from services on other conodes within the cothority network.

In Medchain, we mainly used two services: ByzCoin and Onet service. These services handle client-server communications and enable the user to interact with the conodes. We define Medchain API using these services and later implement the CLI-program on top of this API (see Section 4.3). Table 1 shows some of the most important API calls defined in Medchain as well as resources they take and their responses.

Table 1: Some of Medchain API calls

Name of Method	Description	Resources	Response
CreateQueryAndWait	Spawn a query	User ID, Query definition, Query ID	OK?
UpdateQueryStatus	Update the status of a query	Query ID	OK?
VerifyQueryStatus	Verify the status of a query	Query ID	OK?

To be able to use the services, we need to register them with the default Cothority suite. We use the services to define the API and interact with conodes and contracts.

4.3 App and Command-Line Interface (CLI)

An application, in the context of Onet, is a CLI-program that interacts with one or more conodes through the use of the API defined by one or more services.

We implemented the CLI for Medchain using bcadmin library. The CLI allows the user to start a new ByzCoin blockchain, register new users, initialize and manage pre-developed Darcs, interact with the smart contract, etc. through the command-line. The code for this CLI-app is found in `mc/` directory of Medchain repository.

5 System Evaluation

5.1 Limitations

Medchain has some limitations in its current version. First, it is the low latency in authorizing queries which results in low query throughput. The latency stems from the fact that for query authorization the `InstanceID` of the query should be retrieved so that the authorization is based on the responsible Darc that governs the query. Every time a query is spawned, its `InstanceID` is named so that it is easily fetched from the global state by its name later using function `ResolveInstanceID()` of ByzCoin. However, this function returns the results very slowly and thus impedes the authorization of other queries in the pipeline. Our workaround to overcome this issue was to save the `InstanceID` of all queries in a secondary data type called `QueryKey[]` as soon as they are spawned and every time retrieve the `InstanceID` from `QueryKey[]` instead of the global state. This method helps improve the speed and remove the overhead of transactions needed for naming the query instances at the price of more space complexity. The other drawback of this workaround is that it is prone to error and cannot be trusted.

Second, Medchain CLI requires the admin to define project Darcs by adding rules to them at system startup which is both an involved and time-consuming process. We aim to introduce ways to automate this process and make it more straight-forward.

6 Future Work

In this section, we would like to offer some opportunities for further improving our implemented software solution and the results of this project.

6.1 Integration with MedCo

One of the main purposes and requirements of Medchain is to make it work seamlessly with data service providers, specifically MedCo. However, given that this requirement was not fulfilled in the scope of this project, the full integration of Medchain into MedCo is expected to be done in future versions of Medchain.

6.2 Use of Deferred Transactions

Although we were able to implement deferred transactions in Medchain, we have not been able to test their functionality in this project and we are not sure if they function properly as we have only tested this functionality locally. Therefore, it is expected that in the future this functionality will be tested in a multi-node setup and debugged if necessary.

7 Conclusion

In this project, we designed and implemented a distributed authorization and access management system for medical queries called Medchain. Medchain is written in Go and is based on an existing distributed framework called Cothority and runs on conodes. It enables distributed access management through the use of smart contracts (Darcs) and offers auditability

through the use of permissioned blockchains for recording all the queries it receives over its lifetime. Medchain is mainly designed to work seamlessly with MedCo and overcome MedCo's authorization limitations.

Medchain supports a CLI interface that enables the user to interact with the cluster of servers through the command-line. In future versions of Medchain, we aim to integrate it with MedCo ecosystem and have it work as part of the whole MedCo workflow. We will also improve its front-end and the user interface.

References

[1] Jean Louis Raisaro, Juan Troncoso-Pastoriza, Mickaël Misbach, João Sá Sousa, Sylvain Pradervand, Edoardo Missiaglia, Olivier Michielin, Bryan Ford, and Jean-Pierre Hubaux. Medco: Enabling secure and privacy-preserving exploration of distributed clinical and genomic data. *IEEE/ACM transactions on computational biology and bioinformatics*, 2018.

[2] DEDIS at EPFL. *Cothority Github Repository*, 2020 (accessed January 3, 2020). URL https://github.com/dedis/cothority.

[3] DEDIS at EPFL. *Conode Documentaion*, 2020 (accessed January 3, 2020). URL https://github.com/dedis/cothority/blob/master/conode/README.md.

[4] DEDIS at EPFL. *Skipchain Implementation*, 2020 (accessed January 3, 2020). URL https://github.com/dedis/cothority/blob/master/skipchain/README.md.

[5] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[6] DEDIS at EPFL. *Byzcoin*, 2020 (accessed January 3, 2020). URL https://github.com/dedis/cothority/blob/master/byzcoin/README.md.

[7] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, and Bryan Ford. Certificate cothority: Towards trustworthy collective cas. *Hot Topics in Privacy Enhancing Technologies (HotPETs)*, 7, 2015.

[8] Shawn N Murphy, Griffin Weber, Michael Mendis, Vivian Gainer, Henry C Chueh, Susanne Churchill, and Isaac Kohane. Serving the enterprise and beyond with informatics for integrating biology and the bedside (i2b2). *Journal of the American Medical Informatics Association*, 17(2):124–130, 2010.

[9] Griffin M Weber, Shawn N Murphy, Andrew J McMurry, Douglas MacFadden, Daniel J Nigrin, Susanne Churchill, and Isaac S Kohane. The shared health research information network (shrine): a prototype federated query tool for clinical data repositories. *Journal of the American Medical Informatics Association*, 16(5):624–630, 2009.

[10] David Froelicher, Patricia Egger, João Sá Sousa, Jean Louis Raisaro, Zhicong Huang, Christian Mouchet, Bryan Ford, and Jean-Pierre Hubaux. Unlynx: a decentralized system for privacy-conscious data sharing. *Proceedings on Privacy Enhancing Technologies*, 2017 (4):232–250, 2017.

[11] JBoss. *Open Source Identity and Access Management For Modern Applications and Services*, 2020 (accessed January 13, 2020). URL https://www.keycloak.org/.