Laboratory For Data Security (LDS)

MedChain: A Distributed Authorization 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

Thursday 25th June, 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. 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 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, both authentication and authorization are central which introduces a single point of failure within the network that can compromise the whole network. Second, each participant in MedCo cluster runs its own access management process. Also, there is no distributed immutable logging of queries received in MedCo.

In this project, we aim to improve and fully implement "MedChain: a distributed authorization system for medical queries" that overcomes the drawbacks present in MedCo. MedChain works seamlessly with MedCo and complements its workflow by handling the access management in a distributed manner. MedChain offers auditability as it records all the medical queries from the users in an immutable distributed ledger, i.e., the blockchain.

Contents

1 Introduction

1.1 Problem Definition and Objectives

In this project, we aim to implement a distributed authorization system for medical queries, called MedChain that was designed and partly implemented in a previous project. We also intend to make this system specifically compatible with the ecosystem of MedCo and integrate it with it. MedCo is an existing technology that enables privacy-preserving analysis and sharing of medical data. In MedCo, authentication and authorization are centralized; we aim to integrate MedChain into MedCo in order to enable distributed authorization and avoid single point of failure introduced by the centralized system. Furthermore, in MedChain, we introduce distributed immutable logging of system transactions which allows auditability of activities and queries and can thus be used to prevent malicious activity or data misuse.

1.2 Threat Model

In this section, we explain the adversaries that we considered during system design and implementation. For every adversary, we consider the scope for their capabilities as well the risks they incur to the system.

MedChain is responsible for distributed authorization of medical queries submitted to MedCo and is thus part of a MedCo network. MedCo network consists of a clinical research network that includes several clinical sites organized in a decentralized federation. Clinical sites provide databases that are maintained and monitored separately. There are also investigators who use MedCo to explore medical data [2]. Before authenticated investigators can access the data, they need to be **authorized by MedChain**. Hence, the following adversaries are considered:

Clinical sites: In general, clinical sites are assumed to provide correct information. However, sites do not necessarily trust each other.

Regular user/investigator: This could be a typical user, such as an authorized nurse or physician at a clinical site's research department, who accesses the clinical data and is assumed to be honest-but-curious (HBC) adversary.

(Tech-savvy) investigators: This includes authorized investigators as well as black-hat or white-hat hackers who are assumed to be malicious-but-covert (MBC) adversaries as they may try to DOS the system or infer data by performing multiple queries of data.

1.3 Requirements

MedChain should allow distributed authorization of MedCo queries based on defined and agreed multi-signature rules, i.e., authenticated users should be able to query certain sensitive information only after the access management rules have been met. Furthermore, while checking the authorizations of users for their queries, MedChain should keep an immutable and distributed record of all queries submitted to MedCo so that the system can be audited and malicious activities, attacks, or data leakage can be identified and prevented. Last but not least, as part

of MedCo ecosystem, MedChain should be able to work seamlessly with all systems that function within MedCo ecosystem, most prominently, MedCo-connector that orchestrates MedCo queries.

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

As it was mentioned before, MedChain should be able to perform authorization checks in a distributed manner. To achieve this, we used collective authority in MedChain. Collective authority (cothority) available at [3] 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. Servers that provide Cothority framework, services, and protocols are referred to as **cothority** nodes or **Conodes** [4].

2.1.1 Conode

MedChain is a customized Conode capable of performing desirable tasks. This is achieved through definition and implementation of MedChain services and protocols in a general Conode. This is further explained in Section 3 and Section 4.

2.2 ByzCoin

ByzCoin [5] is an open decentralized blockchain system based on Byzantine fault tolerant (BFT) consensus algorithm. The **Omniledger** paper [6] describes the protocol that ByzCoin implements. ByzCoin uses the Skipchain [7] as its underlying data-structure for storing distributed ledger of keys and values.

Skipchain holds the instances of ByzCoin Smart Contracts (we can consider a smart contract as a class and its instance as an object of this class). Actions that affect these instances result in updates in Skipchain. Instances play an important role in MedChain: each of the queries MedChain receives and tries to authorize is an instance of a smart contract defined specifically for MedChain (this is more explained in Section 3 and Section 4).

In ByzCoin, every instance is stored with the following information in the global state:

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

Below, we explain some important ByzCoin concepts and terminology that are later referred to in this report.

- **Instructions**: It is a request to update the ledger and is created by the client. It can be a:
  - `Spawn`: create a new instance
  - `Invoke`: call a method of an instance such as `update`
  - `Delete`: remove an instance

  An accepted instruction results in a `StateChange`.
- **ClientTransaction**: A *ClientTransaction* holds a collection of instructions and is sent to conodes in the roster for execution.
- **StateChange**: The execution of *ClientTransaction*s results in 0 or more changes in the global state referred to as *StateChange*.
- **Distributed Access Right Controls (Darcs)**: Darcs define rules to access available resources (such as contract instances) in ByzCoin. Darc plays a pivotal role in MedChain as the main entity that enables authorization. Please refer to Section 2.2.1 to learn more about Darcs in MedChain.

2.2.1 Distributed Access Right Controls (Darcs)

In ByzCoin, Darcs are used to manage access to contracts/objects and can thus be used to handle authorizations. Each Darc is a data structure that holds a set of rules as action-expression pairs, i.e., a set of identities (i.e., public keys) in form of an expression is pared with a certain action. Rules must be fulfilled so that the instructions on the instances governed by the Darc can be executed. 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`. Rules defined in Darcs 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 access right definitions.

Every instruction sent to an instance must resolve the rule in the governing Darc. This means that 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 by considering the relevant rules in the Darc governing the instance. If the rule is fulfilled, ByzCoin uses the `InstanceID` to look up the responsible contract for the instance and then send the instruction to that contract for execution.

The authorization process 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).

2.3 MedCo

MedCo [1] is a system that enables sensitive medical data to be explored and shared in a privacy-preserving manner. MedCo guarantees data privacy using collective encryption provided by collective authority [8]. It aims to distribute trust among various medical data providers (such as clinical sites) 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 [9] is a clinical research platforms and together with SHRINE [10] they enable clinical data exploration; (ii) UnLynx [11] allows distributed and secure data processing in MedCo.

In MedCo, user authentication is provided by Keycloak which authenticates users using OpenID Connect as the identity provider. In the following sections, we will provide more details about how MedChain enables distributed authorization and works with MedCo.

3 Design and System Architecture

3.1 General Overview

In this section, we look at the architecture of MedChain system. We first begin by considering the requirements that our software solution should be able to fulfil. Then, we discuss how our solution satisfies the requirements and objectives.

3.2 Requirements

The main goal of MedChain is to offer distributed access management mechanisms for MedCo. We started to design and implement MedChain last semester in another project. The main goal of this project is to improve MedChain software already implemented, develop a docker-based implementation of MedChain and integrate MedChain into MedCo described in Section 2.3. To achieve this, Medchain has to fulfil the following tasks and requirements:

3.2.1 Authorization in Medchain

The goal of authorization is to control the user access to resources (e.g., sensitive medical data) based on agreed rules. In MedChain, we use Darcs (described in Section 2.2.1) to enable user authorization and access management.

3.2.2 Distributed Immutable Logging in MedChain

In MedChain, we use blockchains to offer distributed auditability. The blockchain framework we use in MedChain is ByzCoin (see Section 2.2). Every query sent to MedCo by the user as well as its status, whether it is authorized or rejected, is recorded in the immutable ledger that is accessible at every MedChain node in the network.

3.3 Medchain High-level View

Figure 1 shows how a MedChain node looks like. Every MedChain node is a customized Conode (see section 4 for further details) and has one or more projects defined in it. Projects specify user access controls and define data resources. Each project contains three important building blocks. Below you can find more about projects and their building blocks:

- **Projects**: specify rules for users to access data resources using access right controls.
- **Smart contracts**: define data, in this case key-value pairs of the query (i.e., the key) and its status (i.e., the value), that is stored in distributed ledger
- **Darcs**: define and manage user's access to resources. Every project is attached to a single Darc that determines user access rules for the database governed by the project.
- **Blockchain**: system audit. It is possible to retrieve a record of all queries submitted to MedChain for authorization to check the content of query and the querier's identity.

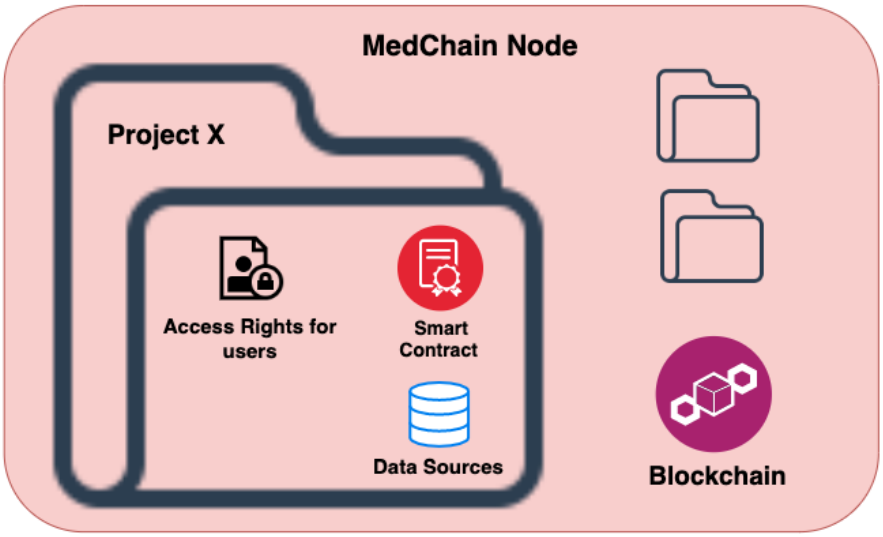

Figure 1: The structure and building blocks of a MedChain node.

3.4 Flow of Queries in Medchain

Figure 4 illustrates the full architecture of MedChain and its detailed workflow starting from query creation and submission by the client until the query is executed. The numbers used in the figure correspond to the steps described below. Also, parts outlined with dashed-line represent the contributions done in this project (further explained in Sections 4, 5, and 6).

1. **User Login**: Client receives the token (i.e., its identity) from Keycloak (using OpenID Connect).
2. **Query Creation & Submission**: User spawns a query. The query is submitted to MedCo-connector (i.e., query orchestrator). Figure 2 shows the simplified structure of a MedCo-connector (explore) query.

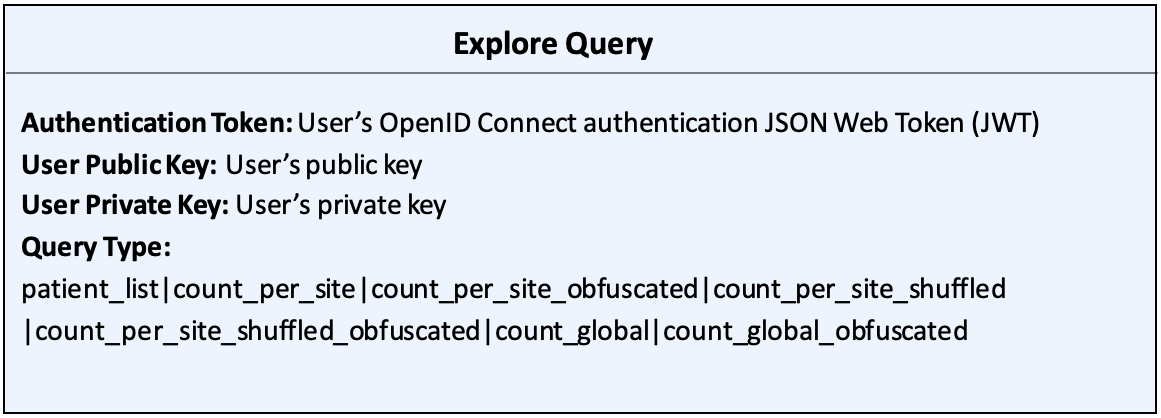

Figure 2: Simplified structure of a MedCo-connector (explore) query.

3. **User Authentication**: MedCo-connector verifies the token (i.e., the identity) of the user and authenticates it.
4. **Submission of Query To Medchain**: MedCo-connector receives the user identity and user query. It then sends the query that contains the user's identity (i.e., part of the token) to MedChain server for authorization. The query that MedChain receives is a key-value pair data structure shown in Figure 3:

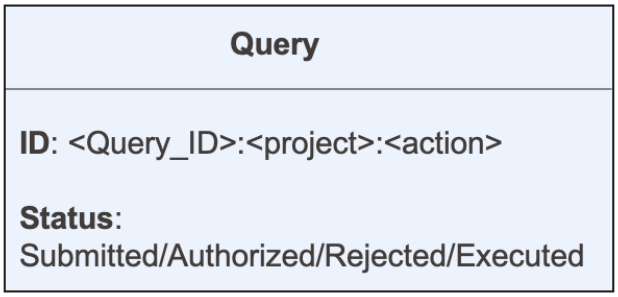

Figure 3: MedChain query data structure

Where `<Query_id>` contains part of the user's token provided by Keycloak, `<project>` is the name of the project (i.e., the database) the user is querying, and `<action>` 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"`, `"Rejected"`, or `"Executed"` .

5. **Query Transaction Creation**: MedChain retrieves the ID o the Darc based on the name of the project and creates a normal (i.e., not deferred, see later) transaction to spawn a `Query` (i.e., an instance of `MedChain Contract`) and signs it. The transaction looks like below:

```
CreateTransaction(byzcoin.Instruction{
    InstanceID:    byzcoin.NewInstanceID(<Project X Darc ID>),
    Spawn: &byzcoin.Spawn{
        ContractID: MedChainContractID,
        Args: byzcoin.Arguments{
                {
                    Name:  <Query_id>:<projectX>:<action>,
                    Value: []byte("Submitted"),
                },
        },
})
```

6. **Query Authorization**: MedChain retrieves the Darc governing the instance and checks the user authorizations for the `<action>`. Depending on the result of authorization checks one of the below scenarios happens:

   - If the Darc rules **are not met** (i.e., user is not allowed to query `<action>`), the query is immediately rejected. Consequently, a new transaction is created by MedChain that *updates* query `Status` to `Rejected` and adds the transaction to the ledger. Also, a message containing the new query status is sent back to MedCo-connector in order to notify it of authorization rejection (this is more explained in Section 4 where MedChain API calls are described)
   - If the Darc rules **are met**, the query is authorized. Hence, two new transactions are created by MedChain. The first one is a normal transaction that updates the query `Status` to `Authorized` which is only signed by this MedChain node and is immediately written to the ledger. The second transaction, however, is a *deferred* transaction (see Section 4). By spawning a deferred transaction, MedChain node proposes a transaction to the whole MedChain network i.e., it broadcasts the instance ID of deferred transaction to rest of the MedChain nodes in network. Then, users

at other MedChain nodes need to *sign* the transaction using the instance ID they receive. When the rules of Darc are met (i.e., a threshold number of signatures are added to the proposed transaction), the transaction can be executed by any of the signers (only one execution is allowed for every proposed transaction). It is important to note that even if all users can sign the transaction, it can only be executed once the rules of corresponding Darc are met. Once the proposed transaction is successfully executed, the transaction is added to the ledger and MedChain sends MedCo-connector a message containing the status of query as `Authorized`.

7. **Query Authorization Result**: In the mean time, MedCo-connector is waiting for a reply from MedChain node. Once authorization (step 6) is done, MedChain sends the result (i.e., query status) back to MedCo-connector which is either `Authorized` or `Rejected`.
8. **Query Execution and Committing Final Query Status To The Ledger**: If MedCo-connector learns that the query has been rejected through a message from MedChain node, it will not execute the query. Otherwise, MedCo-connector executes the query and once done, sends a message back to MedChain so that the status of query is updated to `Executed` and is written to the ledger.

The explained workflow and its mechanisms such as MedChain to MedChain communications (i.e., message propagation among MedChain nodes) and API calls have been fully implemented in this project.

4 MedChain Node Implementation

MedChain is developed in Go. Full Medchain code, its documentation as well as instructions on how to run it are available in Medchain Github repository. In this section, we aim to provide details about the code structure of MedChain node. We also provide instructions on how to setup and run MedChain node (i.e., server). In the end, we revisit MedChain query workflow, but this time in the context of its code and implementation.

4.1 MedChain API

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

4.1.1 Smart Contract

In Byzcoin, smart contract defines the data that is added to the ledger, i.e., data written to the ledger is, in fact, instances of smart contract. Also, smart contract gives us the API to manipulate the instances, for example, in order to create a new instance, we can `Spawn` an instance of smart contract and to update the contract instance, we can apply an `Invoke()` to it (this is explained in more details in 2.2).

In MedChain, we define our own smart contract that implements the suitable data structure: the query. We developed the smart contract in Go (MedChain smart contract is located in `services` directory of the main code repository under the name `contract.go`). This smart contract is a simple key-value contract, meaning that every instance of the contract that is

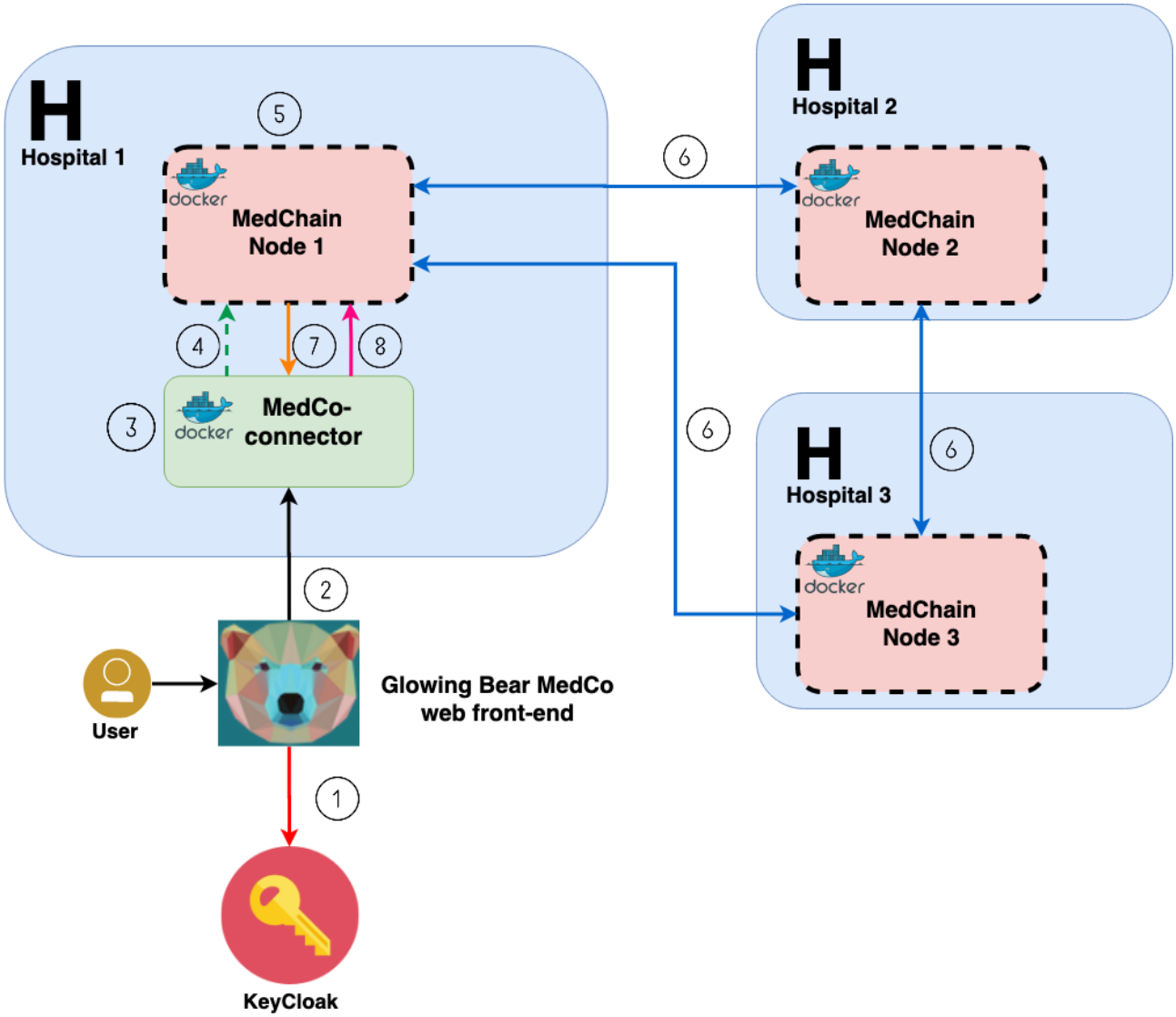

Figure 4: Medchain architecture: workflow of queries in MedChain.

recorded in the ledger, i.e., the query, is a key-value store data structure. MedChain smart contract is identified by its name "**MedChainContract**". This contract is distinguishable from any other contracts used in MedChain deployment such as Darc contracts (see Section 4.1.3) or Value contracts by its ID "**MedChainContractID**". 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 `MedChainContract`. The structure of these queries is shown in Figure 3.

In ByzCoin, it is mandatory that the smart contract 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 `MedChainContract` 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 node 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 determined based on the ID of the Darc contract ruling it. Later, this instance of the contract is retrievable and authorized by the Darc controlling it using this `InstanceID`.

4.1.2 Deferred Transaction

In order to enable multi-signature rules in MedChain, we use ByzCoin *deferred transactions*. Deferred transactions allow a transaction to remain **proposed** (i.e., not written to the ledger) until it receives the threshold number of signatures defined by the Darc governing it. Once it receives enough number of signatures, it can be **executed** and written to 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 `MedChainContract` instance, the following steps are taken:

1. User spawns a `MedChainContract` instance (this is the proposed transaction).
2. User spawns a `deferred_contract` instance with query instance ID as its arguments. In other words, the proposed transaction is the what the deferred instance holds.
3. Signers sign the proposed transaction by invoking an `addProof` on the corresponding deferred instance.
4. User invokes an `execProposedTx` on the corresponding deferred instance to execute the proposed transaction.

It is important to note that ByzCoin does not perform any checks on the signatures added during `addProof` transaction. This means that even garbage signatures can be added to the deferred transactions (but still adding the same signature multiple times is not permitted). However, the signatures and Darc rules are checked during `execProposedTx` transactions. In MedChain, only a maximum of one execution is allowed per proposed query transaction and extra executions are rejected.

4.1.3 Darcs (Distributed Access Right Controls)

Darcs are used to enable authorization in MedChain. In ByzCoin, Darcs are responsible for handling authorization and access management for various resources, such as smart contract instances, and they use action/expression pairs to define rules.

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 Darc data structure. Upon starting a network 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:MedChainContract - "ed25519:<ID_client>"
        --- invoke:MedChainContract.update - "ed25519:<ID_client>"
        --- invoke:MedChainContract.verifystatus - "ed25519:<ID_client>"
        --- _name:MedChainContract - "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:MedChainContract - "ed25519:<ID_client>"` where `ID_client` refers to the ID of the client who is granted the permission for the action `spawn:MedChainContract` which in this case is the ByzCoin admin. 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.

As we mentioned earlier, in MedChain, projects define various databases and in order to control access to the database, every project is associated with a Darc. Here, we consider the example of having Project A and Project B and a cluster of 3 MedChain nodes. We create Darcs for each of these projects. We assume that each project has 3 clients (i.e., one client per MedChain node). 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:MedChainContract - "ed25519:<ID_client1>" |
        "ed25519:<ID_client2>" | "ed25519:<ID_client3>"
        --- invoke:MedChainContract.update -
        "ed25519:<ID_client1>" | "ed25519:<ID_client2>" |
        "ed25519:<ID_client3>"
        --- invoke:MedChainContract.patient_list -
        "ed25519:<ID_client1>" | "ed25519:<ID_client2>" |
        "ed25519:<ID_client3>"
        --- invoke:MedChainContract.count_per_site -
        "ed25519:<ID_client1>" | "ed25519:<ID_client2>"
        | "ed25519:<ID_client3>"
        --- invoke:MedChainContract.count_per_site_obfuscated -
        "ed25519:<ID_client1>" | "ed25519:<ID_client2>" |
        "ed25519:<ID_client3>"
        --- invoke:MedChainContract.count_per_site_shuffled -
        "ed25519:<ID_client1>" | "ed25519:<ID_client2>" |
        "ed25519:<ID_client3>"
        --- invoke:MedChainContract.count_per_site_shuffled_obfuscated -
        "ed25519:<ID_client1>"
        --- invoke:MedChainContract.count_global -
        "ed25519:<ID_client1>"
        --- invoke:MedChainContract.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:MedChainContract.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 signer identities. However, only client 1 is given permission to query

`count_global` from database A (i.e., the action `invoke:MedChainContract.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 `MedChainContractID` of MedChain smart contract, i.e., `MedChainContract`, which is different from the ID of the Darc instance itself. The client must be able to authenticate against a `spawn:MedChainContract` 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: "MedChainContract",
    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. This instance ID is the hook to this instance and the client needs to remember this it in order to manipulate this instance later, for example, invoke methods on it to update its status. 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. Once an instance is named, the client can the name given to the instance ID to retrieve it from the ledger.

After this step, a `MedChainContract` instance spawned by the user (which is in fact the query submitted to MedChain from MedCo-connector) 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.

Important point: since the very first transaction, i.e., the one created right after MedChain receives a query from MedCo, is always immediately written to the ledger (due to auditability purposes) using a `Spawn` instruction of `Value` contract, all project Darcs need to have `spawn:value` and `invoke:value` rules enabled for all Darc users (i.e., using an *at least one* expression).

4.2 MedChain Service

In Cothority, every app (e.g. Command-Line-Interface app) or client (at the front-end) communicates with services to interact with Conodes. In other words, services are responsible for handling Client-to-Conode communications. Services are based on Onet library that provides the overlay network and enables definition of communication protocols in Cothority. A service is created with Conode. It serves client requests, creates protocols for Overlay network (see

later), and handles communication of the information to other services on other Conodes.

In MedChain node implementation, in order to define client-MedChain node communications and enable message sharing among them, we defined our own service. MedChain service code can be found in `services/service.go` in the main code repository. In MedChain service, that is based on Onet, we define services that use Protocols which can send and receive messages. This means that MedChain node uses Onet to send/receive messages to/from the client over the service-API, which is implemented using protobuf over WebSockets. Messages defined in MedChain are explained in next section.

4.2.1 MedChain Messages and API Calls

In MedChain, we mainly used two services: ByzCoin and Onet services that enable client-server communications. We define Medchain API using these services and later implement the CLI-program on top of this API (see Section 5). Messages defined and used in MedChain can be found in `services/struct.go` in code repository. Table 1 shows some of the most important messages defined in MedChain as well as resources they take and their responses.

Table 1: Most Important Medchain API calls

Name of Method	Description	Resources	Response
AddQuery	Spawn a query transaction	User ID, Query definition, Darc ID	Instance ID, OK?
AddDeferredQuery	Spawn a deferred query transaction	User ID, Query definition, Instance ID, Darc ID	Instance ID, OK?
SignDeferredTx	Sign a deferred query transaction	User ID, User keys, Instance ID	Instance ID, OK?
ExecDeferredTx	Execute a deferred query transaction	User ID, Instance ID,	Instance ID, OK?
AuthorizeQuery	Authorize a query	Query definition, Instance ID, Darc ID	Instance ID, Query Status, OK?
GetSharedData	Get instance IDs shared with node	-	Instance IDs
PropageteID	Broadcast instance IDs to roster	Instance ID, Roster	OK?

4.3 MedChain Protocol

In Cothority, protocols are responsible for handling Conode-to-Conode communications. In MedChain, we use the protocol defined in MedCo-unlynx which can be found in `protocols` directory of MedChain repository. The protocol is instantiated with service whenever needed and is used to broadcast the instance ID of deferred transactions to all MedChain nodes in the roster.

Both services' and protocols' binaries loaded into MedChain node at compile time using package imports as it can be seen in main server code found in file `cmd/medchain-server/medchain.go`.

4.4 MedChain Node

MedChain node is a Conode that has specific MedChain services and protocols available in it. The main MedChain node (i.e., server) code can be found `cmd/medchain-server`. Table 2 shows the commands (subcommands of `server`) that are supported in MedChain node.

Table 2: MedChain Server:`server` command and its subcommands

Command	Description	Arguments
`server`	Run MedChain server	`--c`: Server configuration file `--d`: debug-level
`server setup`	Setup MedChain server	-
`server setupNonInteractive`	Setup server non-interactively	`--sb`: server binding `--desc`: node description `--priv`: Private toml file path `--pub`: Public toml file path `--privKey`: Provided private key `--pubKey`: Provided public key
`server config`	Check servers in roster	`--g`: Group definition file `--t`: Set a different timeout

4.4.1 MedChain Node: Setup and Run

The easiest way to setup and run multiple MedChain nodes (locally and as a shell process) is to use the provided shell script `run_nodes.sh` at `/cmd/medchain-server/` directory of MedChain repository. For example, the following code can be executed in order to run 3 MedChain nodes:

```
path/to/medchain/cmd/medchain-server$ go build
path/to/medchain/cmd/medchain-server$ run_nodes.sh -n 3 -d medchain-config
```

The above code, will setup and start 3 MedChain nodes and will put all configuration files in `./medchain-config/`. The file containing public configurations will be in `./medchain-config/group.toml`. The folders created in `./medchain-config/` (i.e., `mc1/`, `mc2/`, ...) each hold private keys of a server in `private.toml`. The servers will then be listening to ports. It is important to note

that each MedChain node uses two ports. MedChain-to-MedChain communication is automatically secured via TLS when you use the unchanged configuration from MedChain server setup. However, MedChain-to-client communication happens on the next port up from MedChain-to-MedChain port and it defaults to WebSockets inside of HTTP.

Additionally, it is possible to run MedChain nodes one at a time and without the mentioned script. To this end, the command below can be used to run a single MedChain node:

```
path/to/medchain/cmd/medchain-server$ go build
path/to/medchain/cmd/medchain-server$ ./medchain-server server setup
path/to/medchain/cmd/medchain-server$ ./medchain-server server setup
```

After the above commands are run, MedChain node is setup. Now, to run the server, the following can be used:

```
path/to/medchain/cmd/medchain-server$ ./medchain-server server setup
path/to/medchain/cmd/medchain-server$ ./medchain-server server
```

Last but not least, one can use Docker to run a single MedChain server or a network of them. This is explained in details in Section 6.

4.5 MedChain Query Workflow Revisited

Now that we have described and discussed the implementation of a MedChain node, we can consider its query workflow with a code and implementation perspective. Figure 5, illustrates various code components of MedChain, operations that happen in MedChain, the interfaces, as well as communications in MedChain. As it is shown in this figure, in the front end, we have the CLI (Command-Line-Interface) client, (see Section 5 for further details) and the Go API client, explained earlier in the section, that interact with MedChain node by sending requests to it. In the back end, MedChain node, also a member of MedChain network, serves the client requests. The steps below explain detailed query workflow in MedChain and correspond to numbered operations in Figure 5:

1. CLI client submits a query to MedChain Go API client for authorization through a client call. The query has the structure shown in Figure 3.
2. Go API client sends the query to MedChain node for authorization per service call. Then, the transaction to spawn the Value contract (that implements a simple key-value pair) instance is **prepared** with the query ID as its key and query status `Submitted` as its value. Please note that the query ID contains what the actual user query is (e.g., patient_list, etc.) and that is the **action** which is authorized or rejected. See Section 3.4 and Figure 3.
3. MedChain looks up the Darc governing the query to check the authorizations for the action.
4. Depending on the Darc rule, the action is either rejected/authorized. Then, the transaction is executed and the status of value query instance is updated correspondingly. Finally, the result is written to ledger after ByzCoin nodes achieve consensus.
5. The status of query is sent back to client through a service call. If the status is `Rejected`, the process stops here.

6. If the query is authorized, Go API sends a request to spawn an instance of MedChain contract through a service call. Its key is the query ID and the status of query is not relevant in this step. A transaction containing the query ID and status (=empty) is created.
7. An instance of deferred contract is **spawned** with the transaction in previous step as its **proposed transaction**. The instance ID of this deferred contract instance is broadcast to all MedChain nodes in the network using MedChain protocol so that their clients can later use this instance ID to sign the proposed transaction.
8. Client sends request to sign the proposed transaction to Go API client. The request contains the instance ID of targeted deferred instance.
9. Go API client invokes an `addProof` on the deferred contract instance holding the proposed transaction in order to sign it.
10. Client sends request to execute the proposed transaction to Go API client. The request contains the instance ID of targeted deferred instance.
11. Go API client invokes an `execProposedtx` on the deferred contract instance holding the proposed transaction in order to execute it.
12. Provided that enough signatures have been already added to the proposed transaction, it is executed and it is added to the blockchain with status `Authorized`.

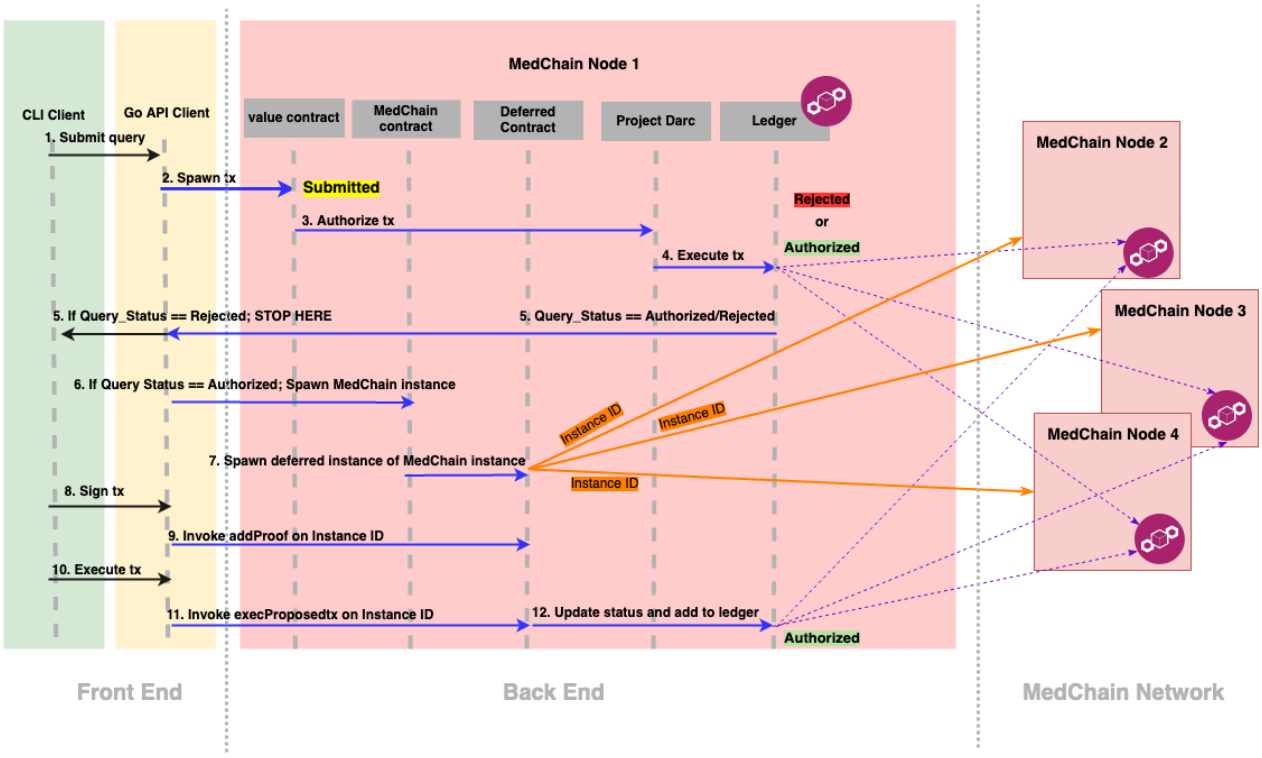

Figure 5: Detailed workflow of queries in MedChain with respect to various code components. In this figure, black arrows show client calls, blue arrows represent MedChain service calls (client-to-MedChain communications), orange arrows show communications handled by the protocol (MedChain-to-MedChain communications), and dashed arrows correspond to ByzCoin-to-ByzCoin communications.

5 MedChain Client Implementation

MedChain supports two ways for clients to connect to the MedChain service: Go API Client and CLI that are explained in the following sections.

5.1 MedChain Go Client

The detailed MedChain Go API can be found in `services/api.go`. Also, examples of Go API usage can be found in `services/api_test.go`.

5.2 MedChain Command-Line-Interface Client

An app, 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 app for MedChain which is a client that talks to the service available in server (MedChain node). The code for this CLI-app is found in `cmd/medchain-cli-client` directory of MedChain repository.

Table 3 summarizes the commands that are supported in MedChain CLI Client. Please note that in the table, `client_flags` refers to:

```
client_flags:
    --bc: ByzCoin config file
    --file: MedChain group definition file
    --cid: ID of the client interacting with MedChain server
    --address: Address of server to contact
    --key:The ed25519 private key that will sign the transactions
```

5.3 How to Use MedChain CLI Client

In order to use MedChain CLI client, a network of at least 3 MedChain nodes should already be up and running. Please refer to Section 4 or Section 6 to learn more about MedChain nodes and how to deploy a network of them. Once a network of MedChain nodes (i.e., MedChain servers) is up and running, the following commands can be used to use MedChain CLI client to interact with MedChain servers, submit a query to the network for authorization and demo query rejection and authorization scenarios.

Please note that all the below commands must be run in `/cmd/medchain-cli-client`.

In order to setup and start ByzCoin ledger, we run the following commands:

```
bash$ go build
bash$ bcadmin --config medchain-config create medchain-config/group.toml | tail -n 1
bash$ bcadmin -c config info
```

We can export below environment variables so that we can more easily use them later:

```
bash$ export BC=medchain-config/bc-xxx.cfg
bash$ export admin1= ed25519:xxx
bash$ export adminDarc=....
```

Table 3: MedChain CLI Commands

Command	Description	Arguments
`create`	Create a MedChain CLI Client	`--client_flags`
`query`	Submit a query for authorization	`--client_flags` `--qid`: The ID of query, `--did`: The ID of project darc `--idfile`: File to save instance IDs
`sign`	Add signature to a proposed query	`--client_flags` `--instid`: Instance ID of query to sign
`verify`	Verify the status of a query	`--client_flags` `--instid`: Instance ID of query to verify
`exec`	Execute a proposed query	`--client_flags` `--instid`: Instance ID of query to execute
`get`	Get deferred data	`--client_flags` `--instid`: Instance ID of deferred data
`key`	Generate a new keypair	`--client_flags` `--save`: File to save key `--print`: Print the private and public key
`darc`	Tool for managing Darcs (see Table 4)	-
`fetch`	Fetch deferred query instance IDs	`--client_flags`

```
bash$ export adrs1=tls://localhost:7770
bash$ export adrs2=tls://localhost:7772
bash$ export adrs3=tls://localhost:7774
bash$ export MEDCHAIN_GROUP_FILE_PATH=medchain-config/group.toml
```

Next, we create keys for users 2 and 3:

```
bash$ ./medchain-cli-client key --save admin2.txt
bash$ export admin2=$(cat admin2.txt)
bash$ ./medchain-cli-client key --save admin3.txt
bash$ export admin3=$(cat admin3.txt)
```

We can run the command below to check the genesis Darc:

```
bash$ ./medchain-cli-client darc show --bc $BC --file
$MEDCHAIN_GROUP_FILE_PATH  --cid 1 --address $adrs1 --key $admin1
--darc $adminDarc
```

In order to create the first client we run:

```
bash$ ./medchain-cli-client create --bc $BC --file medchain-config/group.toml
```

Table 4: MedChain CLI: `darc` Subcommands

Subcommand	Description	Arguments
`show`	Show a DARC	`--client_flags` `--darc`: ID of darc to show
`update`	Update the genesis Darc	`--client_flags` `--identity`: The identity of the signer
`add`	Add a new project DARC	`--client_flags` `--save`: Output file for Darc ID `--name`: Name of new DARC
`rule`	Add signer to a rule or delete the rule.	`--client_flags` `--darc`: ID of the DARC to update `--name`: Name of DARC to update `--rule`: Rule to which signer is added `--identity`: Identity of signer `--type`: Type of rule to use `--delete`: Delete the rule

```
--cid 1 --address $adrs1 --key $admin1
```

Next, we add a default project Darc and call it "Project A Darc". Please note that this functionality is only enabled for test purposes. `medadmin` is the main CLI tool to generate and manage MedChain admin and project Darcs. Please refer to `cmd/medadmin/README.md` for further details.

```
bash$ ./medchain-cli-client darc add --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 1 --address $adrs1 --key $admin1 --save darc_ids.txt --name A
bash$ export projectA=...
```

To create and start clients 2 and 3 we run:

```
bash$ ./medchain-cli-client create --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 2 --address $adrs2 --key $admin2

bash$ ./medchain-cli-client create --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 3 --address $adrs3 --key $admin3
```

Next, we need to add clients 2 and 3 as signers of project A Darc (for demo/test purposes):

```
bash$ ./medchain-cli-client  darc rule --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 1 --address $adrs1 --key $admin1 --id $projectA --name A
--rule spawn:medchain
--rule invoke:medchain.patient_list
--rule invoke:medchain.count_per_site
--rule invoke:medchain.count_per_site_obfuscated
--rule invoke:medchain.count_per_site_shuffled
```

```
--rule invoke:medchain.count_per_site_shuffled_obfuscated
--rule invoke:medchain.count_global
--rule invoke:medchain.count_global_obfuscated
--identity $admin2 --type AND

bash$ ./medchain-cli-client  darc rule --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 1 --address $adrs1 --key $admin1 --id $projectA
--name A --rule spawn:deferred
--rule invoke:medchain.update
--rule invoke:deferred.addProof
--rule invoke:deferred.execProposedTx
--rule spawn:darc --rule invoke:darc.evolve
--rule _name:deferred
--rule spawn:naming
--rule _name:medchain
--rule spawn:value
--rule invoke:value.update --rule _name:value
--identity $admin2 --type OR

bash$ ./medchain-cli-client darc show --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 3 --address $adr2 --key $admin2 --darc $projectA
```

We run the above commands also once for client 3, i.e., using `--identity $admin3`.

In order to submit query, we can use:

```
bash$ ./medchain-cli-client query --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 1 --address $adrs1 --key $admin1 --qid test:A:patient_list
--darc $projectA --idfile InstIDs1.txt
export inst1=$(cat deferred_InstIDs1.txt)
```

Please note that a deferred instance is returned by server if the query is authorized.

To get the deferred query, we can run:

```
bash$ ./medchain-cli-client get --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 1 --address $adrs1 --key $admin1 --instid $inst1

bash$ ./medchain-cli-client get --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 2 --address $adrs2 --key $admin2 --instid $inst1

bash$ ./medchain-cli-client get --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 3 --address $adrs3 --key $admin3 --instid $inst1
```

Furthermore, deferred query instances can be fetched by:

```
bash$ ./medchain-cli-client fetch --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 1 --address $adrs1 --key $admin1

bash$ ./medchain-cli-client fetch --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
```

```
--cid 2 --address $adrs2 --key $admin2

bash$ ./medchain-cli-client fetch --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 3 --address $adrs3 --key $admin3
```

Now, to sign the deferred query we use:

```
bash$ ./medchain-cli-client sign --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 2 --address $adrs2 --key $admin2 --instid $inst1

bash$ ./medchain-cli-client sign --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 3 --address $adrs3 --key $admin3 --instid $inst1

bash$ ./medchain-cli-client sign --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 1 --address $adrs1 --key $admin1 --instid $inst1
```

Finally, to execute the transaction, we use:

```
bash$ ./medchain-cli-client exec --bc $BC --file $MEDCHAIN_GROUP_FILE_PATH
--cid 2 --address $adrs2 --key $admin2 --instid $inst1
```

Last but not least, one can use Docker to run a network of MedChain nodes as well as MedChain CLI client using docker-based deployment of MedChain. This is explained in details in Section 6.

6 MedChain Docker-based Deployment

In order to facilitate the deployment of (multi-node) MedChain network, docker-based deployment of it is enabled that uses docker images built for both MedChain server node and MedChain CLI client.

Docker-based deployment of MedChain is located in `deployment` directory of MedChain repository. Below, is the description of files found in this directory:

- `Dockerfile`: Contains the commands needed to assemble MedChain node docker image
- `client.Dockerfile`: Dockerfile to build MedChain-cli-client docker image
- `docker-compose.yaml`: Multi-container definition of MedChain node and MedChain CLI client
- `docker-entrypoint.sh`: Docker entrypoint script for both MedChain node and MedChain CLI client containers
- `docker-compose-demo.yaml`: Multi-container definition of a network of 3 MedChain nodes and 1 MedChain CLI client (shown in Figure 6) which is also used in the demo.

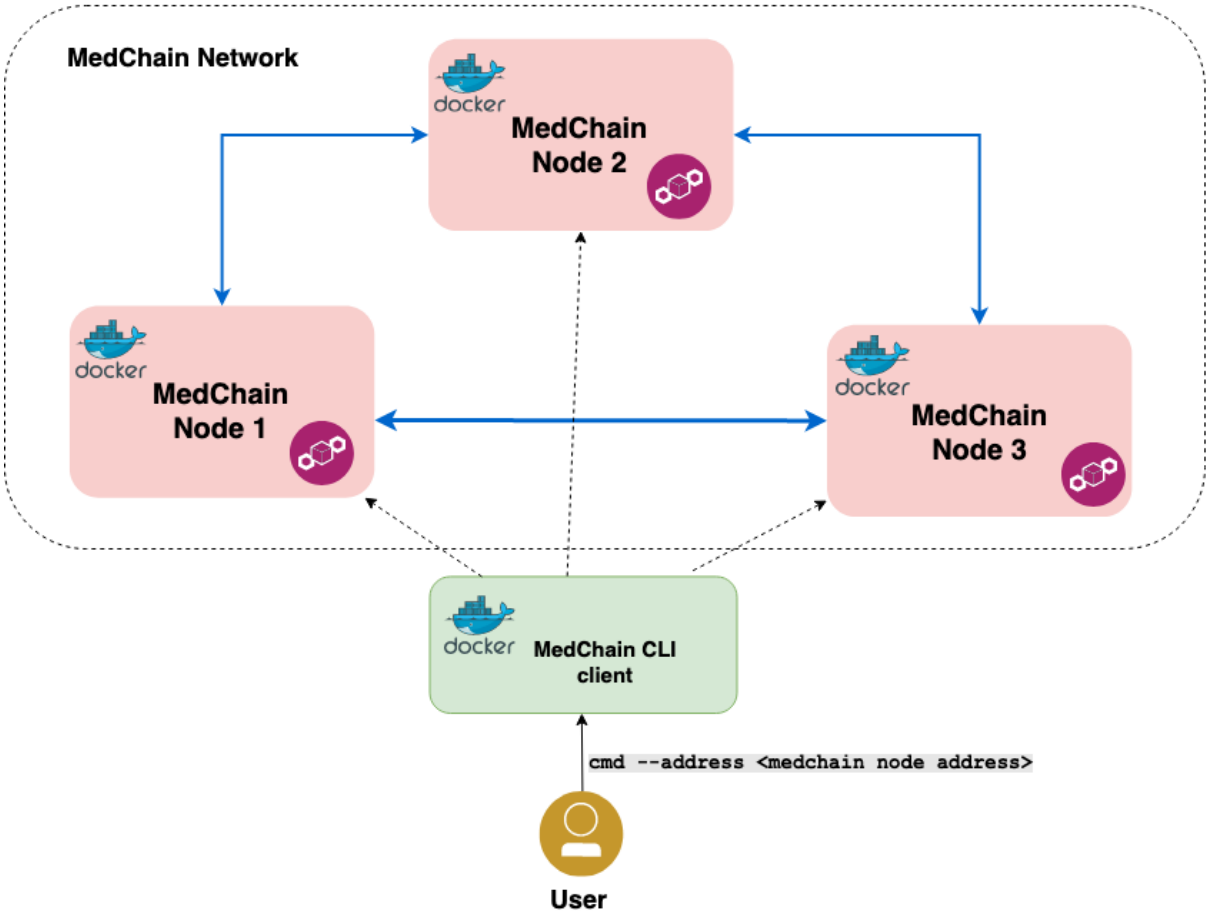

Figure 6: An example of a multi-node MedChain network created using the docker-based deployment of MedChain. This network is defined in `docker-compose-demo.yaml` file in `deployment` directory of MedChain. The user is able to interact with any of the three MedChain nodes (each running as a separate docker container) through `MedChain CLI Client` docker container by providing the address of MedChain node he/she wants to communicate with.

6.1 Use Docker to Run MedChain

Docker can be used to run a MedChain node and its CLI client or a network of multiple MedChain nodes and CLI clients.

To setup and run a single MedChain node and a single MedChain CLI client, and/or to build their docker images, one can run the following commands in `deployment/` directory of MedChain repository:

```
bash$ docker-compose -f docker-compose.yaml up --build
```

This command will build MedChain server and CLI client images, setup and run the server in a container, and create a MedChain CLI client container that can interact with running server. Please note that flag `--build` is only necessary if the user needs to build docker images for MedChain node and its CLI client.

Please note that before we can setup and run MedChain nodes using `docker-compose.yaml` file, we need to have the `private.toml` file of every MedChain node in `deployment/medchain-config/mcX` directory where X corresponds to node index. We can use the script `run_nodes.sh` provided in `medchain-server` folder to setup as many MedChain nodes as we want. For example, to setup 3 MedChain nodes, we can use the command below:

```
bash$ mkdir medchain-config
bash$ ../cmd/medchain-server/run_nodes.sh -v 5 -n 3 -d ./medchain-config/
```

Once all the servers are up and running, we need to use MedChain CLI client and run the commands in the running container, to this end, we can run:

```
bash$ docker exec -it <name_of_cli_client_container> bash
```

We can, then, use the commands described in Section 5 to interact with MedChain node through the CLI.

To setup and run a multi-node network, one can define their own network in a docker-compose file and run it using:

```
bash$ docker-compose -f <docker-compose file path> up
```

Important point: Once the network is up and running, we need to update the `private.toml` file of each node as well as `group.toml` file with the IP address of corresponding docker containers. To get the IP address of each container, we can use:

```
bash$ docker network inspect <name of MedChain network>
```

7 System Evaluation

In this section, first, we explain how we evaluated MedChain's performance. Then, we present the evaluation results and discuss them. Finally, we describe MedChain limitations.

7.1 MedChain Simulations

The simulation package of Cothority (Onet) can be used to run multiple code simulations on localhost, mininet, or deterlab and can be used to evaluate how implemented services and pro-

tocols work. We have defined and used simulations to evaluate MedChain's performance.

We performed experiments on **localhost** in order to see if MedChain system scales well as the number of nodes in the network increases. To achieve this, we measured time the it takes to do various actions on a single query. More details about the evaluations is given in the following section. The code used to run simulations in MedChain can be found in `simulation/` directory of MedChain repository. In order to run simulations and reproduce the results presented here, you can use one of the following commands in `simulation/`:

```
bash$ go test simul_test.go
```

or

```
bash$ go build
bash$ ./simulation service.toml
```

In fact, code in `simulation/service.go` and `simulation/service.toml` can be used to simulate a client talking to MedChain service.

7.2 System Evaluation Setup and Results

In order to evaluate the performance of MedChain system and understand if it scales well with increasing number of participating MedChain nodes in the network, we measured the time it takes to do various actions (see Table 7) on one query in network sizes of 5, 10, 20, and 30 MedChain nodes.

As it was mentioned earlier, simulations were run on localhost. Server CPU used is a 3.1 GHz Intel Core i7 with 16 GB of RAM. We have run 5 rounds of simulations with 5, 10, 20, and 30 nodes (hosts). Table 5, summarizes the setup used in simulations. Figure 7 and Figure 8 show the results of performance testing of MedChain service using simulations averaged over 5 rounds of simulations for a **single** query. The exact measurement values are recorded in Table 6. Finally, in Table 7, we summarize what different actions in Table 6 and plots of results refer to.

Table 5: MedChain Simulation Setup

Number of Simulation Rounds	5
Number of MedChain Nodes	5, 10, 20, or 30
Branching Factor (BF)	2
RunWait	2h
BatchSize	5

Please note that in Table 5, **Branching Factor** corresponds to the number of children each node has. **RunWait** corresponds to how long to wait for a run (one line of .toml-file in which we define simulations) to finish.

7.3 Discussions

As it was mentioned earlier, we measured the time of various actions in MedChain for one query to understand if the service's performance scales well as the network size (number of MedChain

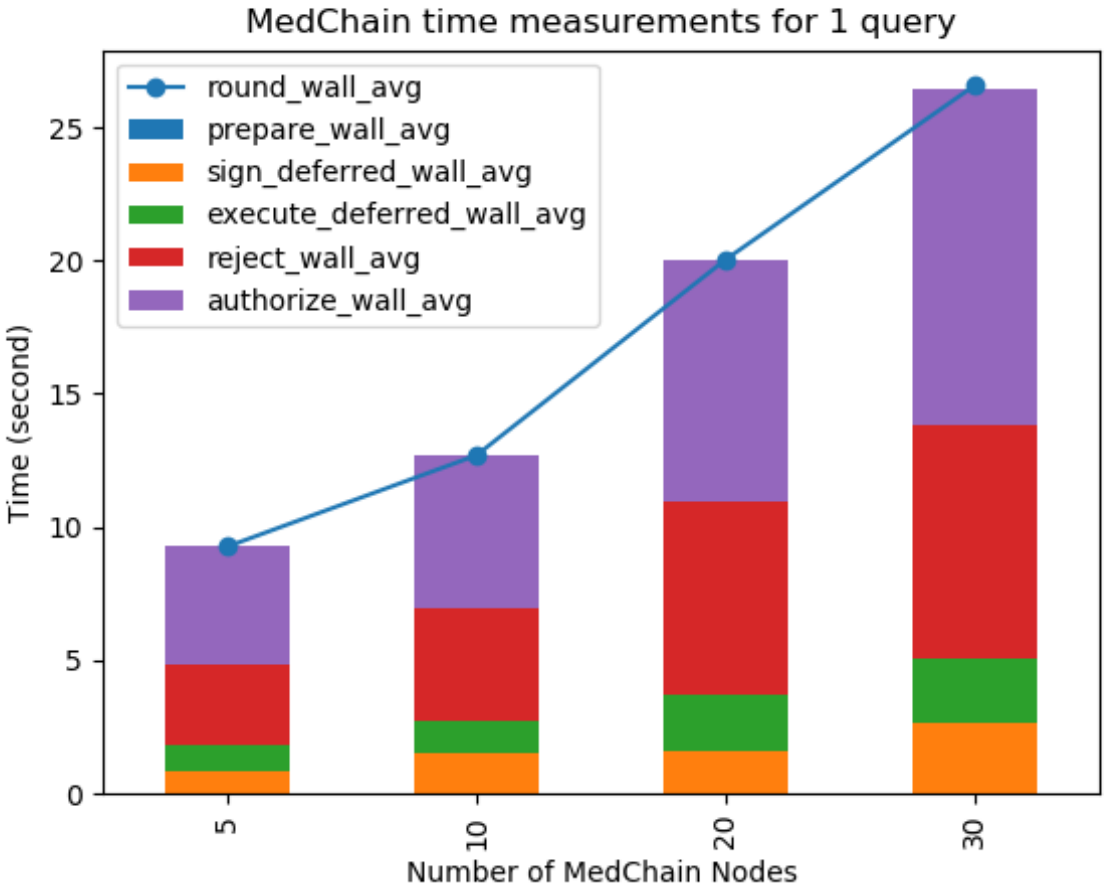

Figure 7: Medchain System Evaluation Results: Average wall time taken by various actions in MedChain (in seconds) vs. number of MedChain nodes in network. Actions are explained in Table 7. Measurement setup is described in Table 5. Please note that *prepare_wall_avg* is so small that it is not depicted here. However, it is visible in plot of Figure 8, where time is shown in logarithmic scale.

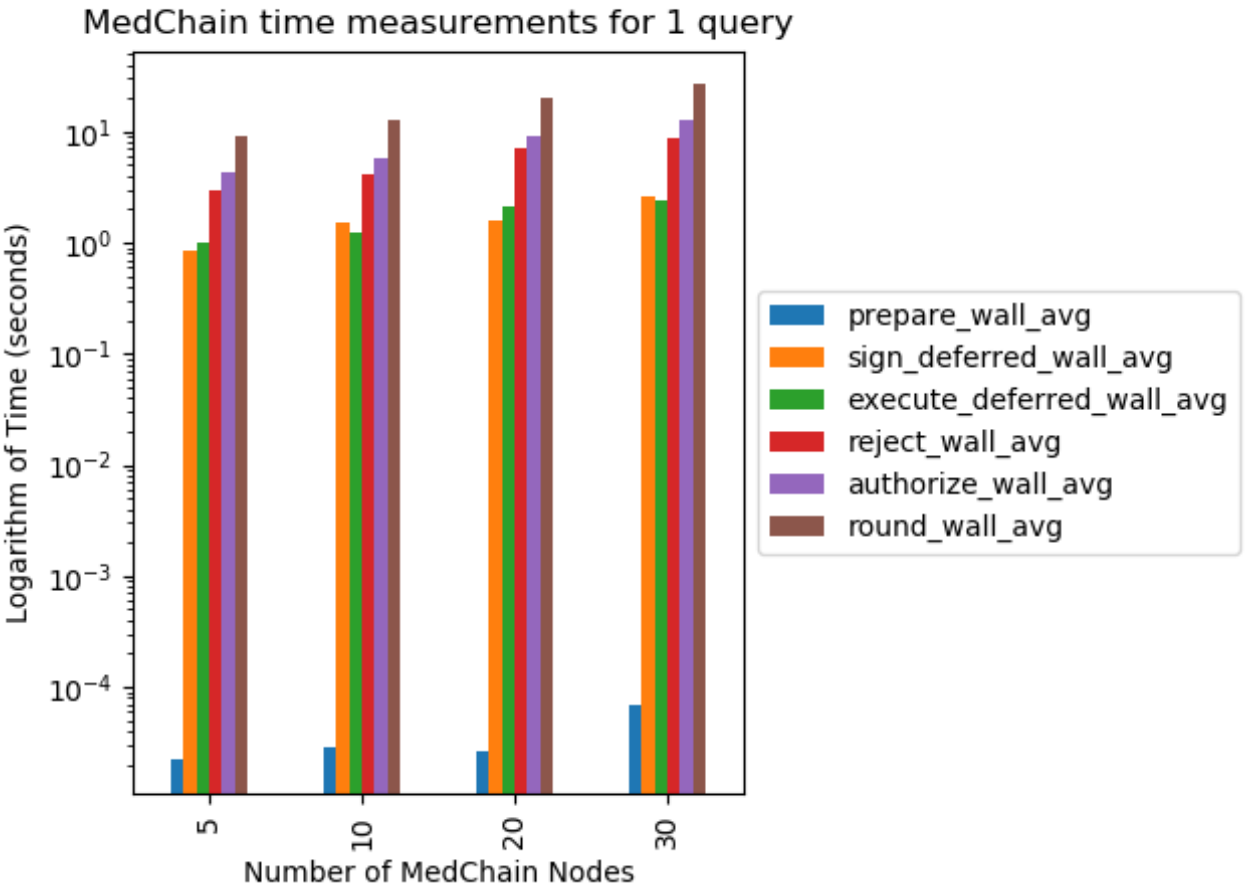

Figure 8: Medchain System Evaluation Results: Logarithm of average wall time taken by various actions in MedChain (in seconds) vs. number of MedChain nodes in network. Actions are explained in Table 7. Measurement setup is described in Table 5.

Table 6: MedChain System Evaluation Results: Time measurements for various actions in MedChain averaged over 5 rounds for a single query

Action	**5 Nodes**	**10 Nodes**	**20 Nodes**	**30 Nodes**
prepare	48 μs	38 μs	34 μs	254 μs
authorize	5.13 s	7.20 s	10.27 s	14.36 s
reject	3.02 s	4.19 s	7.21 s	8.72 s
sign_deferred	0.85 s	1.52 s	1.61 s	2.63 s
execute_deferred	1.00 s	1.24 s	2.13 s	2.45 s
round	9.27 s	12.69 s	20.03 s	26.57 s

Table 7: MedChain System Evaluation Results: Definition of actions measured. Step numbers referred to in this table correspond to steps of MedChain query workflow described in Section 4.5 and illustrated in Figure 5

Action	**Definition**
prepare	Total time it takes to do steps 1 and 2
authorize	Total time it takes to do steps 3 , 4, 5, 6, and 7.
reject	Total time it takes to do steps 3, 4, and 5
sign_deferred	Total time it takes to do step 9
execute_deferred	Total time it takes to do step 11
round	Total time it takes to do steps 1-12 (except for steps 8 and 10)

.

nodes in the network) grows. Simulation results are illustrated in Figure 7 and Figure 8. Please note that in these plots, `xxx_wall_avg`, corresponds to the average time measured using the *wall clock*. Wall clock measurements indicate the amount of time an external observer would have measured for a specific process to complete. So, if the system waits for a reply from the network, this waiting time is also included in the measurement. Please refer to Onet Simulation library for further details about its measurements.

The results are averaged over 5 rounds of simulations using the setup described in Table 5 for a single query.

Plots of Figures 7 and 8 show exactly **the same measurement results**. However, since *prepare_wall_avg* time was very small and was not shown in Figure 7, logarithm of time measurements was used in Figure 8, so that *prepare_wall_avg* is also illustrated. The actions measured and plotted in the figures are explained in Table 7. The exact time measurements are recorded in Table 6.

According to the plots, we can see that, as expected, all processes take longer to complete as the network size (i.e., the number of MedChain nodes increases). Also, for every query, *authorization* in MedChain is the longest process time-wise as it consists of more steps of the query workflow than the other processes (except for the full round); it also includes the time it takes for instance ID to be broadcast to the whole MedChain network which grows with the network size. The other observation is that, *sign_deferred* and *execute_deferred* take almost the same amount of time as they are both ByzCoin transactions on an instance of a deferred contract. However, *execute_deferred* action is likely to take more time than *sign_deferred* since

it checks the signatures added to a proposed transaction as well as the Darc rules while the latter does not perform such checks on signatures while adding them. Furthermore, we observe that *reject* action takes more time to complete than both *sign_deferred* and *execute_deferred* since it includes the time it takes for the ByzCoin to achieve consensus and add transactions to the ledger. As the sum of all actions equals the full round time, we conclude that MedChain service does not introduce time overheads in the workflow.

Last but not least, we observe that the time of all actions increases almost linearly (in some cases sub-linearly) with number of nodes. Thus, we can conclude that the system exhibits scalability as number of MedChain nodes increases. However, it is important to note that, in reality, the time it takes for a query to be authorized by MedChain depends on the project Darc rules and the time it takes for all required users to sign a system deferred transaction so that the Darc rules are met.

7.4 Limitations

7.4.1 Darc Evolutions and Deferred Transaction Concurrency Issue

We already discussed how Darcs are used to handle authorizations in MedChain. Also, in Section 4.1.3, we explained that rules in Darcs can be evolved and are, thus, Dynamic. Therefore, we can consider the case when a proposed deferred transaction of a query instance is awaiting the signature from user A before it can be executed by MedChain; in the meantime and before the query receives enough signatures it needs to be able to get executed, 2 system admins create (deferred) Darc evolution transactions to change the Darc rules governing the subject query instance synchronously. We can imagine that Darc evolution #1 results in the rules for query to change in such as way that user A's signature is no longer needed and thus the query can be executed any time, while Darc evolution #2 requests signatures from new users B and C so that the deferred query transaction can be executed. In this case, the order in which Darc evolution transactions are executed can change the final result state. For instance, if the Darc evolution #1 is accepted first, the query can immediately be executed while this would not be the case with Darc evolution #2 being accepted.

Another scenario in which such concurrency issues can happen is when a a deferred query transaction and a Darc evolution transaction of the Darc that governs the query are created concurrently. Again, depending on the order the two transactions are executed, the query can be rejected or authorized.

This behavior is allowed in ByzCoin due to the fact that deferred transactions, such as the Darc evolution transaction, remain proposed until a threshold number of signatures are added to them and hence the time and order in which signatures are received for concurrent transactions matter. In order to prevent such synchronization issues, we need to implement concurrency control algorithms in either MedChain, or ByzCoin.

8 Future Work

MedChain is designed and developed to work seamlessly with MedCo and it is currently fully ready to become a part of MedCo ecosystem. So, as a future work, MedChain has to be fully integrated into MedCo ecosystem so that the queries received by MedCo-connector are

authorized by MedChain before they are executed by MedCo-connector.

Furthermore, adding a rich user interface for MedChain in addition to the CLI, such as a web front-end, can be considered in order to help the user interact with MedChain more easily. As an example, the user could perform rich searches (by time, user, etc.) in a list of all queries that need his/her signature or execution through the front-end.

9 Conclusion

In this project, we implemented a distributed authorization and access management system for medical queries called MedChain. MedChain is mainly based on a distributed framework called Cothority and its blockchain called ByzCoin and is written in Go. MedChain was designed and partially implemented in a similar project before. However, in this project, we were able to improve its API, most importantly by adding the API calls and interfaces that did not exist in the older MedChain version, as well as its CLI. Consequently, MedChain query workflow is complete at the moment and this means that MedChain is ready to be integrated into other software ecosystems, such as MedCo. Using the CLI, the user is able to submit queries to MedChain node for authorization, sign previously submitted queries that need user's signature, interact with MedChain node to verify the status of query, etc.

Also, a docker-based deployment of MedChain was also developed as a result of this project. Using this docker-based implementation, it is possible to setup and run a multi-node MedChain network using docker images of MedChain node (server) and Medchain CLI client using a single command.

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] Jean Louis Raisaro, Juan Ramón Troncoso-Pastoriza, Mickaël Misbach, E Sousa Gomes de Sá, Joao Andre, Sylvain Pradervand, Edoardo Missiaglia, Olivier Michielin, Bryan Alexander Ford, and Jean-Pierre Hubaux. Medco: Enabling privacy-conscious exploration of distributed clinical and genomic data. In *4th International Workshop on Genome Privacy and Security (GenoPri'17)*, number CONF, 2017.

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.