MedCo: Using permissioned blockchains for privacy-conscious sharing of distributed medical data

Riccardo Succa and Niccolò Sacchi

Supervisors:
Jean Louis Raisaro, Juan Troncoso-Pastoriza
Linus Gasser, Eleftherios Kokoris Kogias, João André Sá

École polytechnique fédérale de Lausanne

June 12, 2017

Abstract

To unlock the potential of data sharing we need solutions that improve data security and, in particular, accountability. In this report, we describe the problem of providing accountability over privacy-conscious medical data sharing in a federation of medical institutions that may not trust each other. Then, we discuss the design and implementation of a solution whose main goal is to enable accountability. The latter must be provided over operations performed by entities belonging to the system. We designed and implemented four services to manage four categories of information: Topology, Identity, Access Control and Query Logging. These four services are implemented on top of the skipchain [8] technology and will be integrated with the UnLynx framework [9]. It is worth noticing that the modularity of the proposed solution makes it possible to adapt services to other similar scenarios involving different types of sensitive data.

I Introduction

i Problem Statement

The need of protecting sensitive medical data while still being able to share them across different medical institutions, that do not necessarily trust each other, is now more urgent than ever. The new legal landscape set forth in the European General Data Protection Regulation (GDPR) imposes strong requirements both on the side of confidentiality (avoiding undue leakages and restricting access to data) and accountability (lawful and privacy-conscious processing auditable by external and internal entities, breach notification, etc.). The implementation of a secure ecosystem would reduce the probability of sensitive data disclosure and therefore increase the number of hospitals that are willing to securely share their medical data. A possible solution is to provide a central server trusted by all the medical institutions that would manage all the sensitive data. However, the trusted server would be a central point of failure that would need strong security protocols to protect it. Trust decentralization is a key feature to achieve the goal of sensitive data sharing and effective collaboration among different hospitals that do not necessarily trust each other. Moreover, records must be stored while guaranteeing their transparency and integrity. Institutions, users, patients among others expect to be able to audit the actions undertaken by other parties, e.g., who accessed which information, which policies were adopted and if they were breached, etc., so to control there are no misbehaving entities. An assortment of collected logs that can be accessed by authorized auditors would allow to (1) determine if the results of the processes have been correctly produced and (2) determine that data have been correctly protected during the process and after the release of the results. A Data Protection Officer (DPO), hospitals in the federation or even the patients themselves could act as independent auditors for the actions performed on the personal and identifiable data. If such an ecosystem were empowered with accountability then there would be more institutions and patients willing to share medical information which would encourage and foster medical research. Several research groups

[2] employed distributed ledger technologies to address accountability. Yet, this challenge is still far from being solved.

ii Solution Overview

In this report, we propose a new solution to provide distributed accountability in the above-mentioned ecosystem. Our solution distributes trust among different independent servers. It is based on a permissioned distributed ledger which ensures transparency and strong integrity. In particular, our design relies on the skipchain technology, implemented by the Dedis lab at EPFL [8]. Our solution complements an already existing protocol that guarantees strong data confidentiality and privacy protection, (i.e. UnLynx) [9].

iii Contributions

Our design provides an efficient distributed solution to accountability when sharing sensitive data. We underline the fact that the design is indeed flexible and can be seamlessly adapted to fulfill other requirements and enable accountability in other scenarios.

iv Outline

In Section II, we explain which entities are involved in the system and which are their roles. In Section III, we discuss the hypotheses under which system is assumed secure. In Section IV, we describe the details of the design of the proposed solution. In Section V, we explain the techniques and tools employed in the implementation of the solution. . In Section VII, we discuss how our work improves on the state of the art. Finally, in Section VIII, we draw our conclusions, summarizing our achievements and future steps.

II System Model

The system comprises different entities to enable secure and privacy-preserving data sharing, as shown in Figure 1.

- The *skipchain cothority* is a group of independent servers, referred as conodes, in a distributed peer-to-peer network. They run the services to manage and maintain the skipchains.
- The *data cothority* has a central role in the ecosystem. It consists of a set of independent servers in a distributed peer-to-peer network. Its main task is to handle incoming queries and compute their respective results without disclosing patient identities nor compromising their privacy.
- *Patients'* medical information is stored encrypted or unencrypted in the different *data providers*. The medical data is stored depending on the consent expressed by the patients, who can grant or deny the storage and sharing of their own medical information.
- The *institutions* represent organizations (Universities, Laboratories, Hospitals, etc.) that want to access and/or share medical data. Institutions can also be data providers.
- The *users* affiliated to an institution have access to the system and can send queries to the data cothority. The latter forwards queries to the data providers so to collect medical information and then compute the result.
- Finally, *auditing authorities* may be used to detect and react to possible breaches of the system, e.g. attempts of patients' de-anonymization, users' misbehaviors, servers' malfunctions among others. Nonetheless, a protocol to enable accountability and logs recording is imperative for auditing authorities to achieve their task.

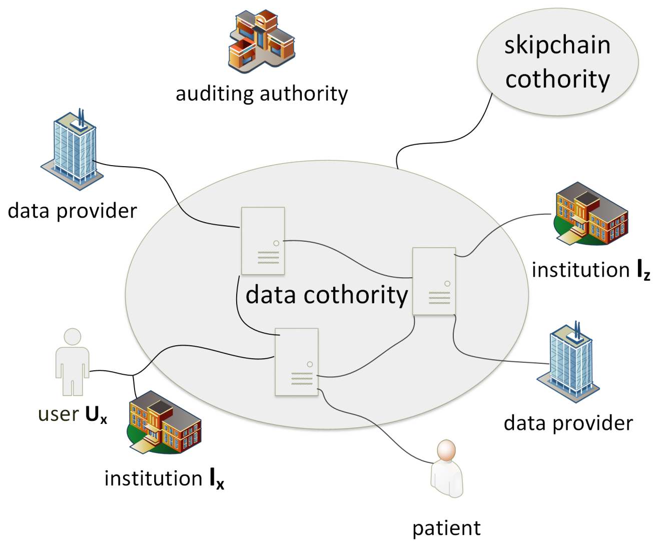


Figure 1: *The system comprises one or more skipchain and data cothorities, medical institutions, data providers and, possibly, an auditing authority.*

III Threat Model

Entities are considered to behave as follows:

- The Byzantine Fault Tolerance (BFT) consensus running on the *skipchain cothority's* servers, and

used to manage the skipchains, allows at most one-third of breached servers.
- The *data cothority* is considered to have a majority of non-malicious servers. As long as only a minority of servers is compromised, the system works properly.
- *Data providers* are trusted to correctly store and supply the encrypted or unencrypted medical data.
- *Users* and *institutions* are considered honest-but-curious. They provide correct medical information but could try to infer sensitive information by performing a series of targeted queries to de-anonymize the patients' information.

We note that protection of data confidentiality at rest and during computation is out of the scope of this work and it can be already covered by the existing system UnLynx. The latter empowers practitioners, at medical institutions, with federated and privacy-preserving queries on sensitive medical data.

IV Proposed Solution

We identified four classes of information that must be stored, therefore leading, for the sake of versatility, to the implementation of four independent services:

- *Topology service*: Stores and retrieves the series of system topologies that followed in time. It is used both to retrieve which data providers can be contacted to supply data and to determine, at a specific time, which data processors were involved in the computation of a result.
- *Identity service*: Identities of medical institutions, physicians and doctors are stored in order to both authenticate them and be able to bind an action to its agent.
- *Access Control service*: Access rights policies of each medical institution must be recorded and publicly known so that it is possible to determine which classes of medical data, provided by a medical institution, can be accessed by users belonging to a specific category or department.
- *Query Logging service*: Every time a query is sent, it must be stored, whether it is actually performed or dropped by the data cothority due to insufficient rights of the user. Thus, it is possible to reconstruct the chain of actions undertaken by each user and determine if they behave maliciously.

The four services are implemented on top of the skipchain technology and are run by the skipchain cothority. The skipchain implements several novel features that make it fast and efficient, e.g. the use of signed forward pointers let it possible to scan fast though a skipchain, while a Byzantine Fault Tolerant (BFT) consensus algorithm enables high scalability and low transaction latency.
The conodes of the skipchain cothority must import the four services in order to correctly run them when they are started.
The division of the services not only reflects the different classes of information, but also brings some other advantages. It is easier, for example, to restrict which services can be accessed by a certain entity and which operations it can perform. Moreover, the division allows us to better manage and update the code and also distribute the workload over different skipchains: when the system need to update a certain type of information, only the related skipchain is updated leaving the others untouched.
The accountability property is achieved by adding a time field in each skipblock of the skipchains so that it is possible to retrieve the data of a particular moment in time. The integrity property of skipchains guarantees that skipblocks are not modified. Furthermore, the services not only enable accountability, but are also used for authentication and authorization processes and other tasks that are discussed in the following sections.
We implemented the system so that each service provides an API used to interact with the services running on the servers of the skipchain cothority. When a client calls an API function, a message is sent to a server of the skipchain cothority and managed by the corresponding service.
Data must be verified and collectively signed by the data cothority before being sent to the skipchain cothority. Conodes of the skipchain cothority, before creating and appending the skipblock containing those data, run a verification function to establish the correctness of the signature. The collective signature of the data cothority servers is performed by the CoSi protocol [5] that requires a predetermined threshold of servers of the data cothority to participate. The verification function receives the signed data and the set of the public keys of the data cothority servers that signed the data. The signature is verified with a key computed from the aggregation of all those public key (Schnorr signature).
On the other hand, requests to only read certain skipblocks may not need to be signed nor controlled by the data cothority. Indeed, for transparency, every institution should be able to access every piece of information stored in the skipchains. However, the protocol could

be changed to restrict reads of the skipchains to be executed only after after approval of the data cothority (i.e. signing the request with the CoSi protocol and making the skipchain cothority control the signature before revealing any information).
In the following subsections, we describe the design of the four services.

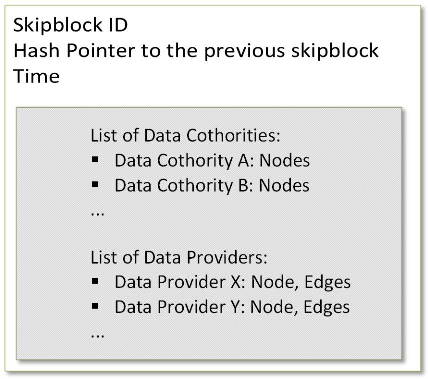
i Topology Service

i.1 Purpose and Design

The main goal of the Topology service is to store the network topology of the system and to keep track of its changes.
The service is based on one skipchain, where each skipblock contains the entire state of a particular moment. When a change in the topology is triggered, a new topology state is computed, collectively signed by the data cothority and sent to the skipchain cothority; the latter adds a new skipblock to the skipchain after verifying the signature. Any entity can retrieve the current state of the system by reading the last skipblock of the skipchain. Entities can use this information to know which data processors and data providers are in the system and where to send or receive data.

i.2 Stored Data

The topology is represented as a graph, where each entity (i.e. data processor or data provider) is a node and the connection between two entities is an edge. Each node contains the necessary information for its localization inside the network, in particular, its IP address, its name, and a description.


Skipblock ID
Hash Pointer to the previous skipblock
Time

List of Data Cothorities:
- Data Cothority A: Nodes
- Data Cothority B: Nodes

...

List of Data Providers:
- Data Provider X: Node, Edges
- Data Provider Y: Node, Edges

...

Figure 2: *Contents of a topology skipblock*

Figure 2 shows the information stored inside the skipblock.

- *Data cothority*: for each data cothority, we save the list of the servers that belong to it. In a typical situation, the system has only one data cothority, but we enable the possibility of having more, for example, a backup system could be needed when the main data cothority is under maintenance. Inside the skipblock, the data processors can be sorted by importance, so if the first one is not reachable, the entities can use the next one.
- *Data provider*: each data provider is a node in the system. We also store the edges connecting it to one or more nodes of the data cothority which represent the servers trusted by the data provider. A data provider only replies to requests coming from one of its trusted servers. In this way, the data providers do not have to trust the entire data cothority, but only a subset of it.

i.3 Functionality

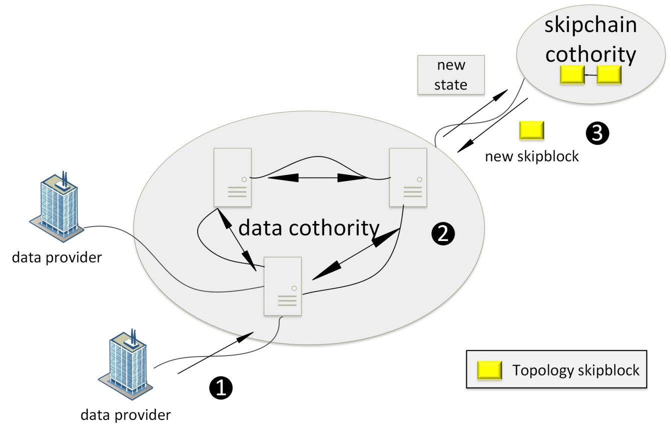


Figure 3: *A data provider joins the system: (1) data provider sends a request to join the system, (2) the request is collectively signed, (3) the new state is inserted in a new skipblock of the topology skipchain*

Figure 3 shows the case where a data provider requests joining the system.

1. The data provider submits an application to the data cothority (or an administrator) for joining the system: this step is possibly done offline with a contract between the parties.
2. A server of the data cothority computes the new state. The latter is then collectively verified and signed by the data cothority.
3. Finally, the new skipblock is sent to the skipchain cothority, which verifies the signature, updates the Topology skipchain and returns the new skipblock.

ii Identity Service

ii.1 Purpose and Design

The Identity service has a central role in the management of users' information. In order to be able to perform any operation in the system (in particular send queries) the user must be enrolled in the Identity service and provide the required information. When a user sends a request, the service is used to verify his/her identity and establish whether he/she has the rights to perform that request.
To create a flexible service, we decided to design the identity service as a hierarchical structure, as shown in Figure 4. It is divided into three main levels:

- *Identity skipchain*: this is the main skipchain that contains the list of all the institutions registered in the service. The institutions are associated to their respective institution skipchain. During the enrollment of a new institution, the latter is inserted along with the link to its Institution skipchain which can be provided directly by it or can be created at the moment of registration.
- *Institution skipchain*: each institution manages a skipchain containing the list of all the users under its control. Again, each user is associated to a structure that contains all the necessary user information, provided at the moment of enrollment. In particular, the user must give their *user skipchain* and the institution adds the group which the user belongs to. In fact, each institution identifies a number of groups with different rights, and each user must be assigned to one of the groups (see the Access Control service for more details).
- *User skipchain*: this skipchain is provided during the registration phase and contains all the public keys relatives to the devices associated to the user. The management of this skipchain is under the control of another service [6].

This hierarchical structure enables distributing the insertion of new skipblocks in the skipchains. For example, the users enrolling or leaving an institution trigger updates only in that institution skipchain without affecting the others. In the same way, when a user needs to append a skipblock to his/her skipchain, the others' skipchains are not modified. This can be an advantage with respect to having all the information stored in only one skipchain, considering that, in a system with several institutions, the users could be added or removed frequently and that the skipchains are retrieved every time a user must be identified. Another advantage is that a hierarchical structure prevents storing too much information inside only one skipchain so our design makes the size of the skipblocks more controllable leading to better scalability of the service. As in the other services, accountability is achieved by adding the time field in every skipblock. By looking at the different levels, it is possible to establish, for example, which devices a user had registered at the moment of a particular query, which group he/she belonged to and if he/she had the correct rights, and other information.

ii.2 Stored Data

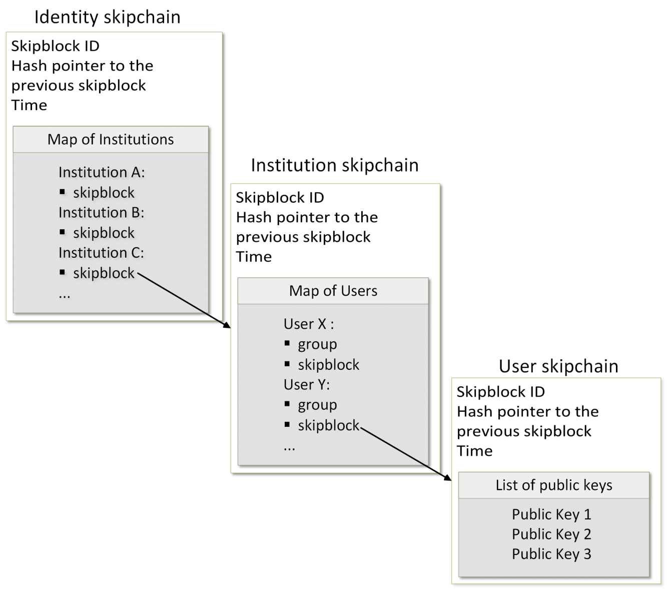


Figure 4: *Identity service hierarchical structure and contents*

Every skipblock of the identity skipchain contains a map linking every institution with one of its skipchain's skipblock, which can be used as an entry point to retrieve information of the institution's skipchain. The institution skipchain contains a map linking every user affiliated with that institution to the user information, i.e. their group and a skipblock of their skipchain. Note that these maps are hash maps, thus insertions and deletions are efficient (O(1) complexity).

ii.3 Functionality

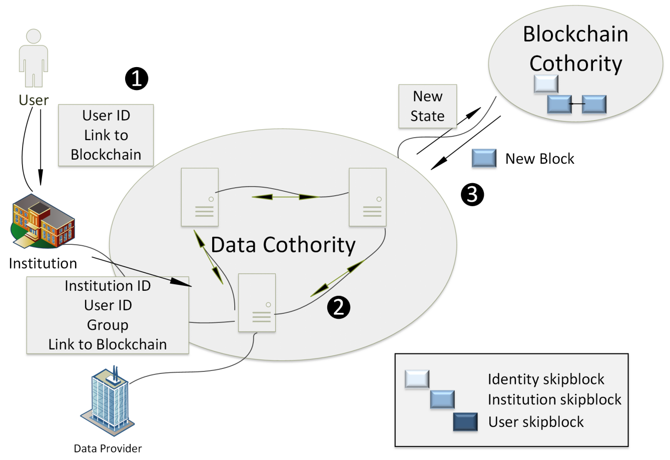


Figure 5: *A User enrolls in the system: (1) the user and the institution provide the necessary information, that is (2) collectively signed by the data cothority. (3) The skipchain cothority adds a new skipblock to the Institution skipchain*

We can explain how the service works by analyzing the simple case when a new user wants to join the system (Figure 5).

1. The user is enrolled in an Institution, providing the link to his personal user skipchain. The institution gives also the information about the user's group.

2. The request is sent to the data cothority, that inserts the new user in the map and signs the new state. The latter is then sent to the skipchain cothority.

3. The skipchain cothority verifies the signature and adds a new skipblock to the Institution skipchain, where the user is present. The new skipblock is then returned.

Analogous operations are done when an institution wants to join/leave the system, this time triggering an update in the Identity skipchain.

iii Access Control Service

iii.1 Purpose and Design

The Access Control (AC) service is used to store and update the access rights policies of the institutions. A set of groups, to which users will be assigned, is defined a priori and fixed for the whole system. Each institution defines its own access rights policy by assigning to each group a set of rights. Each user affiliated with an institution is then assigned to a group, thus binding them to the set of actions allowed by their institution. It is worth noting that the group to which a user belongs is bound to their identity and therefore managed by the Identity service. This service is used in two scenarios:

- To authorize a user to perform a certain action. E.g. when a user sends a query, the data cothority must check if the user's institution allows he/she to perform that query by checking its access rights.
- To enable accountability. It may be necessary to check which were the access rights policies at a specific point in time so to verify if all the actions were legitimately performed.

The AC skipchain keeps track of all the changes in the institutions' policies. Every time an institution changes its policies, the new information must be signed and appended to the skipchain so that its new policy is publicly known to all the other entities.

iii.2 Stored Data

For each institution, every skipblock contains a map with the group as key and a set of rights as value, as shown in Figure 6.

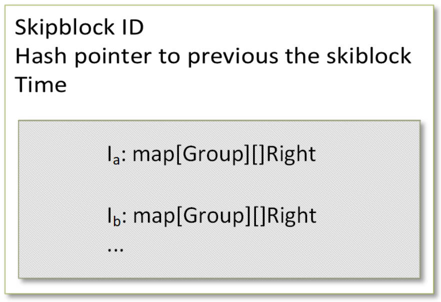


Figure 6: *AC skipblock contents*

iii.3 Functionality

The following use case depicts the sequence of steps undertaken when an institution updates its access rights policies, as shown in Figure 7.

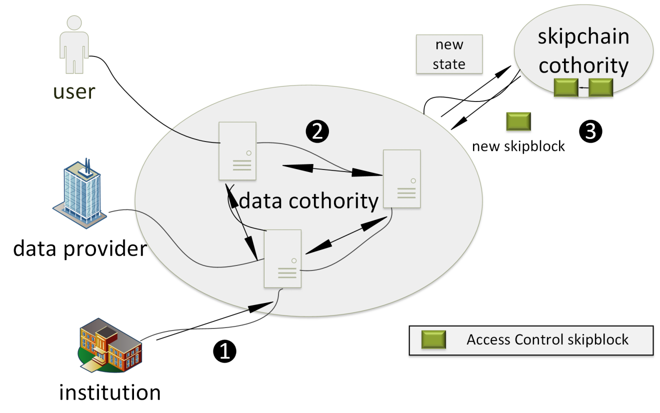


Figure 7: *An institution changes its rights policy: (1) the institution sends a request to the data cothority to update its rights policies, (2) the data cothority verifies the request and (3) updates the AC skipchain*

1. The institution sends a message with its new access rights policy.
2. The data cothority retrieves the latest skipblock of the AC skipchain and computes the new skipblock. The latter is verified and collectively signed by the data cothority.
3. Then, the skipblock is sent to the skipchain cothority. which verifies the signature and appends it to the AC skipchain.

iv Query Logging Service

iv.1 Purpose and Design

The Query Logging (QL) service's main objective is to enable accountability over queries performed by users. Since this service is mainly accessed when a user requests the execution of a query, it must work in real-time to avoid that the user waits too long before the result of his/her query is returned. In our system, whenever a query is sent by a user, the data cothority verifies that the user has both the rights to perform it and enough privacy budget. The rights are checked by accessing the aforementioned Access Control service. The privacy budget is used to bound the number of queries that a user can perform and is needed to guarantee differential privacy. Since the latter is updated every time a query is performed by a user we decided to store the users' privacy budgets in the QL skipchain so that only this skipchain is updated, once every time a query is performed. If the user does not have either the rights or enough privacy budget, then the attempt to perform a non-allowed action is recorded and the query dropped. Otherwise, if the user passes both the tests then the data cothority executes the query by sending it to the data processors and computing the result following UnLynx' privacy-preserving protocols. This result is in the form of a number representing the amount of patients that satisfy the query criteria. Furthermore, random noise is obliviously added to the result to ensure differential privacy. It is worth noticing that the same noise must be added whenever the same query is performed; otherwise, the querier could perform the same query several times and average the obtained results, thus obtaining a good estimate of the exact number of patients that satisfy the query criteria. Query, timestamp, user identifier, encrypted result and encrypted noise must be either stored in a central database or locally by the servers of the data cothority before the result is returned to the user. A hash of all the information related to the query is stored in the QL skipchain as its footprint. Moreover, the QL skipchain also stores a proof of the noise that *should* have been added to a query. The storage of proofs not only cuts down the possibility of data disclosure but also reduces the amount of data stored in a skipblock, therefore reducing the amount of time required to sign it and append it to the skipchain. Afterward, it will be possible to retrieve the performed queries and also verify they were not corrupted by computing their hash and comparing it with the one stored in the skipchain. However, we not only want to retrieve the noise that was added to a result but also verify it is indeed the correct noise (we recall that two equivalent queries must be added the same noise). To efficiently manage the noise that must be added to a result at a specific point in time we decided to use a Merkle tree whose leaves are lexicographically sorted by the encryption of the result. Therefore, every leaf binds to a specific result and would store a pointer to a previously performed query with that result. The Merkle tree is used in two scenarios:

- The data cothority, while computing a result of a query, uses the Merkle tree to check if there was a previously performed query that returned the same result and retrieve the same noise. If no query returned the same result, then the data cothority computes a new noise for the current query and updates the Merkle tree with the new information.
- Entities can check the correctness of noise added to a query at time t_i by comparing it with the noise retrieved from the Merkle tree of time t_i.

The Merkle tree, like the queries, can be stored locally or in a database, while its root is stored in the skipchain as its footprint. The data cothority can retrieve the Merkle tree and use the Merkle root stored in the skipchain to verify it has not been corrupted. At last, since the number of skipblocks equals the number of stored queries, it is advisable to store and index the queries in the same order as their proofs are stored in the skipchain. To this extent, the hash of the ith query would be stored in the skipblock with index i. This service not only enables the possibility to audit the chain of queries performed by users but also to check if those queries were correctly managed by the data cothority.

iv.2 Stored Data

Every skipblock, as shown in Figure 8, contains the following information:

- A map with the user identifier as the key and the privacy budget as value. It is used to keep track of the current privacy budget for every user. It is checked every time a user sends a query and updated every time the query is actually performed.
- A Merkle tree root, used as a footprint of the Merkle tree. It is used by entities to check the Merkle tree has not been corrupted.
- The hash of the requested or performed query and its related information. The correctness of the hash of a query is checked every time an entity needs a proof that shows the query has not been corrupted.

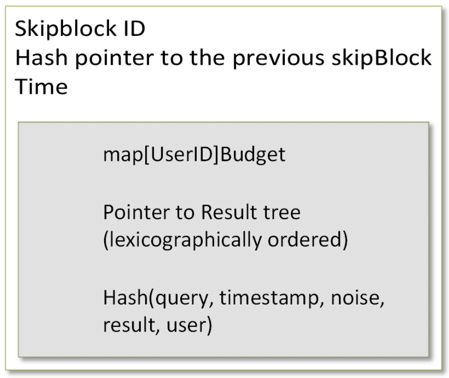

Skipblock ID
Hash pointer to the previous skipBlock
Time
map[UserID]Budget
Pointer to Result tree (lexicographically ordered)
Hash(query, timestamp, noise, result, user)

Figure 8: *QL skipblock contents*

iv.3 Functionality

The following use case shows the step-by-step flow of a user which performs a query, as shown in Figures 9 - 10 - 11.

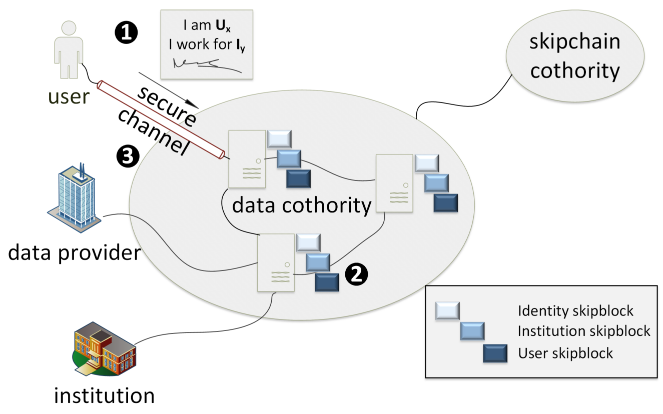


Figure 9: *(1) The user logs-in. (2) The data cothority verifies the credentials and (3) opens a secure channel.*

1. User U_x, affiliated with an institution I_y, logs-in with his/her credential to open a secure channel with the data cothority.

2. The data cothority uses the identity service to assess U_x is an employee of I_y and to retrieve retrieve U_x's group and public keys so to verify his signature.

3. The secure channel is finally opened.

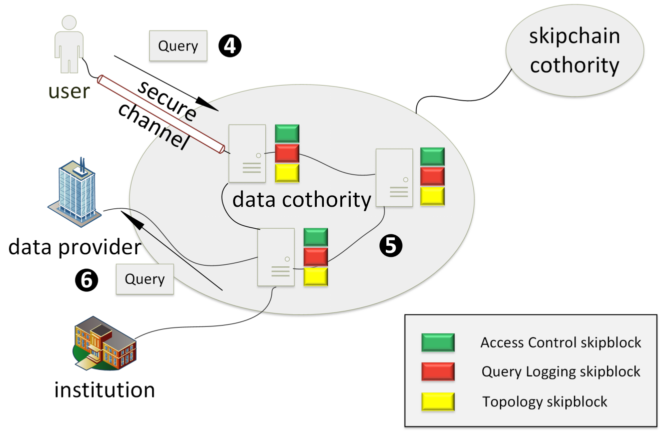


Figure 10: *(4) The user sends the query. (5) The data cothority checks the user's rights and privacy budget and (6) sends the query to the data providers.*

4. User U_x sends the query through the secure channel.

5. The data cothority checks the user U_x has the rights to perform the query and enough privacy budget. Moreover, every server retrieves the topology to locate data providers.

6. The query is sent to all the data providers.

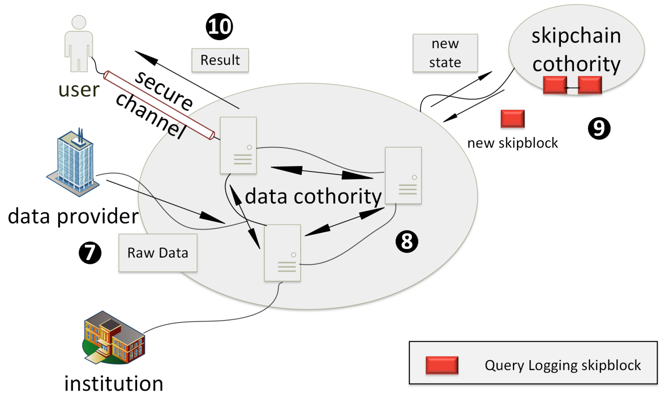


Figure 11: *(7) The data providers supply the data. (8) The data cothority retrieves/computes the result and the noise and (9) updates the QL skipchain. (10) Finally, the result is returned to the user.*

7. The data providers supply the requested data.

8. The data cothority computes the result (UnLynx) and checks whether a previous query returned the same result. If so, it then retrieves the same noise, otherwise, computes a new noise and updates the Merkle tree. Then, the new skipblock, containing the hash of the query and of its related information, the privacy budgets of the users and the Merkle tree root, is computed, verified and collectively signed by the data cothority.

9. The new skipblock is sent to the skipchain cothority which verifies the signature and appends it to the QL skipchain.

10. Finally, the result is returned to the user.

V Implementation

All the services are implemented in Go and are composed of two parts: the APIs, accessed by clients, and services, implemented on top of skipchain [8] and run by servers of the skipchain cothority. APIs and services communicate through messages that are handled by the *Onet v1.0* [7] protocol. Each message sent by the API to a server of the skipchain cothority is handled by the related function of the service. When a new service is implemented, it has to be imported by conodes of the skipchain cothority, so that each of them runs it. When a conode starts, it will run all the imported services. Whenever a skipchain is updated, the latest skipblock is returned to the client that requested the update. The client must store at least the ID of one skipblock of the skipchain they are interested in. Every time a client wants to perform an operation on a skipchain, it has to provide the ID of a skipblock belonging to that skipchain.

i Topology Service API

- *NewTopologyClient*: Initializes and returns a Client object which is used to properly call the following AC APIs.
- *CreateNewTopology*: Requests the creation of a new Topology skipchain, where the initial state of the system is passed as parameter. The request has to be signed by the data cothority servers with the CoSi protocol. The skipchain cothority verifies the signature before creating the Genesis-Block, containing the initial data. This skipblock is returned to the caller.
- *GetLatestTopology*: Receives as input a skipblock belonging to the Topology skipchain and fetches the latest skipblock of that skipchain.
- *UpdateTopology*: Requests the update of an existing Topology skipchain. Receives as input a skipblock belonging to the Topology skipchain that has to be updated and a StateTopology object representing the information, already signed by the data cothority, to be appended to the skipchain. The function is used either when a new entity joins or leaves, or the graph of the network changes. If no errors occur, the newly added skipblock is returned.
- *GetTopologiesByTime*: Given a Topology skipblock and a time interval, retrieves all the skipblocks belonging to that skipchain whose data was created in that interval of time. It is used for accountability purposes, to retrieve the set of data providers and data processors that were operative in a certain interval of time.

ii Identity Service API

As in the other services, all the functions that create/update skipchains must be signed by the data cothority and verified by the skipchain cothority.

- *NewIdentityClient*: Initializes and returns a Client object which is used to properly call the following AC APIs.
- *CreateNewIdentity*: Requests the creation of a new Identity skipchain. No data is contained in the genesis skipblock. When institutions join the sys-

tem, the data contained in the latest skipblock is updated.

- *CreateNewinstitution*: Requests the creation of a new Institution skipchain. Receives as input the identifier of the institution and the UserMap (this parameter can also be Null) which maps each user to their skipchain. The institution is not inserted in the Identity skipchain yet, the function only allows it to create its own skipchain.
- *GetLatestIdentity*: Receives as input a skipblock belonging to the Identity skipchain and fetch the latest skipblock of that skipchain.
- *GetLatestinstitution*: Receives as input a skipblock belonging to the Institution skipchain and fetch the latest skipblock of that skipchain.
- *AddInstitution*: Requests the update of the Identity skipchain. Receives as input the skipblock of the Identity skipchain that has to be updated, the institution identifier and the skipblock of the Institution skipchain. Therefore, an institution must create its skipchain before calling this function. The map stored in the latest skipblock of the Identity skipchain is updated with the new information. The latter is then verified and collectively signed by the data cothority before being appended to the Identity skipchain.
- *RemoveInstitution*: Requests the removal from the Identity skipchain of the institution whose identifier is passed as parameter. Receives as input the skipblock of the Identity skipchain from which the institution must be removed. The skipchain of the institution is not deleted but is kept stored by the skipchain cothority. In this way, if the institution re-joins the system, it can give the previous skipchain avoiding to create a new one.
- *AddUser*: This function allows to add a new user under an institution. It requires as parameter a skipblock of Institution skipchain, the user identifier and a structure containing the user's skipchain and the group to which they belong. The user's skipchain contains the public key of their devices and it is created and managed by the user. The data cothority updated the information contained the latest skipblock of the institution's skipchain, collectively sign the new skipblock and send it to the skipchain cothority that will append it to the skipchain.
- *RemoveUser*: When a user leaves an institution, the latter deletes him or her from its skipchain. The request is sent to the skipchain cothority, that will add a new skipblock where the user is no more present.
- *UpdateUserGroup*: An institution may also want to update the group relative to a user. This function changes the user group with the new one passed as parameter and triggers a new skipblock to the skipchain.
- *GetIdentityByTime/GetinstitutionByTime*: This two functions work in the same way but on different skipchains, the first on the Identity skipchain, the second on the Institution skipchain. Given a skipblock and a time interval retrieves all the skipblocks belonging to that skipchain whose data was created in that interval of time. It is used for accountability purposes, either to retrieve which institutions belonged to the system or to which users were employed by an institution.

iii Access Control Service API

- *NewAccessControlClient*: Initializes and returns a Client object which is used to properly call the following AC APIs.
- *CreateNewAccessControl*: Requests the creation of a new AC skipchain. It gets as input a DataAccessControl object which represents the data to be stored in the genesis skipblock. The data must be signed with the collective key of the data cothority before being stored in the skipchain.
- *UpdateAccessControl*: Requests the update of an existing AC skipchain. Receives as input a skipblock belonging to the AC skipchain that has to be updated and a DataAccessControl object representing the information to be signed and appended to the skipchain.
- *GetLatestAccessControl*: Receives as input a skipblock belonging to the AC skipchain and fetch the latest skipblock of that skipchain.
- *GetAccessControlByTime*: Given an AC skipblock and a time interval, retrieves all the skipblock belonging to the AC skipchain whose data was created in that interval of time. It is used for accountability purposes to check which were the access rights policies of the institutions at a specific point in time.

iv Query Logging Service API

Recall that it is up to the client to properly manage and store the Merkle tree, noise, and the queries. The client uses this service to store the proofs in a public and transparent way.
API description:

- *NewQueryLoggingClient*: Initialises and returns a Client object which is used to properly call the following QL APIs.
- *CreateNewQueryLogging*: Requests the creation of a new QL skipchain. It gets as input a DataQueryLogging object which represents the data to be stored in the genesis skipblock. The data stored in the skipchain must be signed with the collective key of the cothority before being appended to the skipchain.
- *UpdateQueryLogging*: Requests the update of an existing QL skipchain. Receives as input a skipblock belonging to the QL skipchain that has to be updated and a DataQueryLogging object representing the information to be signed and appended to the skipchain.
- *GetLatestQueryLogging*: Receives as input a skipblock belonging to the QL skipchain and fetch the latest skipblock of that skipchain.
- *GetSkipblockQueryLogging*: Given as input a skipblock and an index, retrieves from the QL skipchain the skipblock with the given index. It is used when an entity wants to retrieve the hash of a query as a proof that it has not been corrupted.
- *GetQueryLoggingByTime*: Given a QL skipblock and a time interval, retrieves all the skipblocks belonging to the QL skipchain whose data was created in that interval of time. It is used for accountability purposes, to retrieve the proofs of queries performed in a certain period of time.

VI Evaluation

We used Mininet [1] to simulate a realistic virtual network between servers. Each server ran on a separate machine and was connected to the others by a 1Gbps link with a communication delay of 10ms. For each of our servers, we used machines with two Intel Xeon E5-2680 v3 CPUs with a 2.5GHz frequency, 256GB RAM that supports 24 threads on 12 cores. We set, as initial setup, 3 servers for both the data cothority and the skipchain cothority.

Three operations are sequentially executed after the creation of the (random) data to be stored in the skipchain: (1) firstly, the CreateTopology() function is called to create the topology skipchain, then, (2) the latter is updated with an UpdateTopology() and, finally, (3) the latest skipblock is fetched. We recall that data has to be verified and collectively signed by the data cothority before being stored in the skipchain, i.e. during CreateTopology and UpdateTopology calls. The server of the skipchain cothority which received the request to create a skipchain or append a new skipblock sends the received data the root node of the data cothority (a random server), which starts and coordinates two phases sequentially:

- *Agreement phase*: the root server broadcast the data to all the other servers of the data cothority. Each server verifies the correctness of the topology (e.g. if every servers is assigned at least to a data provider and if the IP addresses exist and are reachable), hashes it with an answer (accepted/rejected) and returns the answer with the signed hash to the root server. The root node controls that every server accepted the data. For this test, we set the threshold of servers that must accept the data to be the 100%.
- *CoSi phase*: then, the root server employs the CoSi protocol to request the servers to sign the hash of the data. Again, all the servers have to participate.

The signed data is then returned to the server of the skipchain cothority which initially requested the verification. This server creates the skipblock after verifying the signature with the collective public key.

The performance evaluation consists in measuring the time needed to create and update the topology skipchain and to fetch the latest skipblock while modifying three parameters, i.e. size of the skipblocks (Figure 12), number of servers (Figure 13) and the latency (Figure 14). Moreover, in the last graph (Figure 15) we show the time it takes to complete each phase of the protocol, i.e. agreement phase (divided in communication and verification of the data), CoSi phase and update of the skipchain.

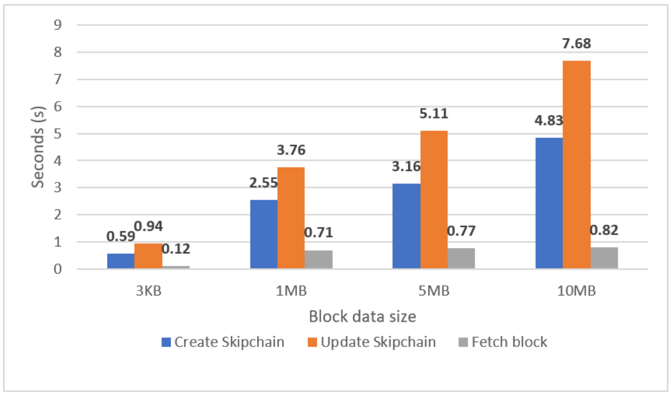


Figure 12: *Time required to create the skipchain, append a skipblock and fetch the latest skipblock while variating the skipblock size.*

The first graph (Figure 12) shows the runtime of the three operations with respect to data sizes of 3KB, 1MB, 5MB and 10MB. The fetch of the latest skipblock is reasonably scalable. The creation and update times grow with the size of the data (the larger is the skipblock, the higher is the time to sign and to send it). However, thanks to algorithms that exploit concurrency, those times grow slower than the amount of data.

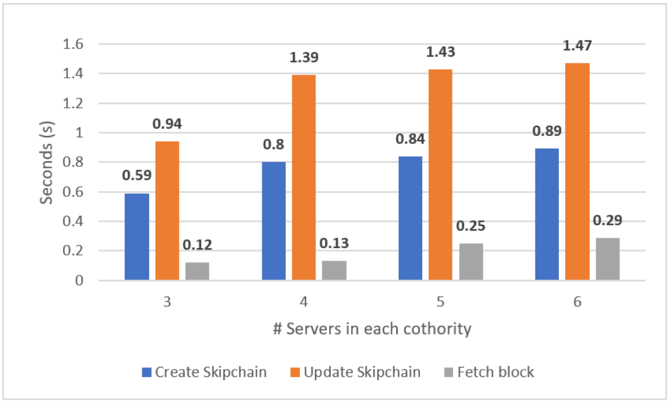


Figure 13: *Time required to create the skipchain, append a skipblock and fetch the latest skipblock while variating the number of servers.*

From the second graph (Figure 13) we can notice that all the times slightly grow with the number of servers, evidence of the scalability of the protocol. However, for a stronger proof of scalability further tests should be undertaken employing a larger amount of servers.

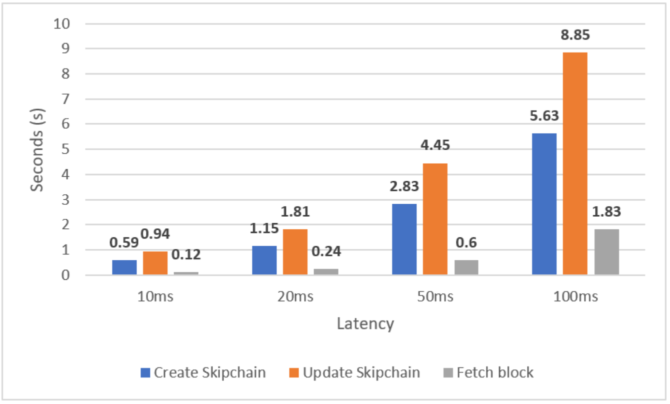


Figure 14: *Time required to create the skipchain, append a skipblock and fetch the latest skipblock while variating the delay.*

The third graph (Figure 14) shows the runtime of the three operations with respect to different latencies, i.e. 10ms, 20ms, 50ms, 100ms. As expected, latency heavily influence all the times and the operations show an almost linear dependency with latency. In a real environment we should expect 10ms of latency.

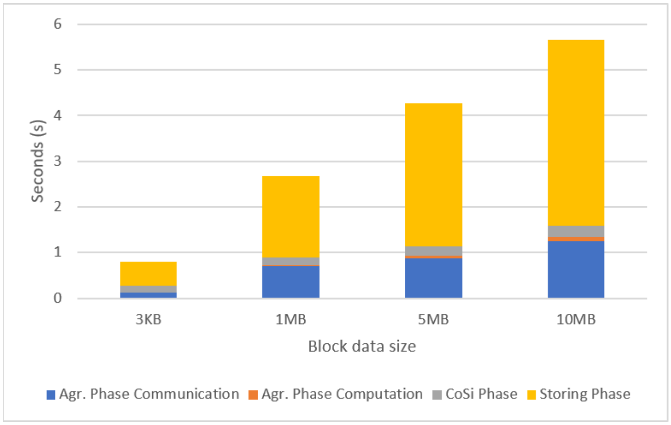


Figure 15: *Update skipchain time divided into its components: agreement phase, CoSi phase and storage of the new skipblock.*

The last graph (Figure 15) shows the components of the update skipchain operation. The time is divided into three parts: agreement phase, CoSi phase and, storing phase (i.e. the phase in which the signature is verified, the skipblock is created and appended to the skipchain by the skipchain cothority). The communication part of each phase, which includes the marshaling and unmarshaling of the data, is what takes most of the time. We might be able to optimize the functions and achieve a higher performance.

We note that the agreement and CoSi phases used in the simulation are not efficient nor fault-tolerant. In a real environment, it is certainly more reasonable to either lower the threshold or employ a more efficient protocol, e.g. the BFTCoSi [4], a Byzantine Fault Tolerant Collective Signing which implements PBFT by having first a commit-round, followed by a signing-round. For this reason, our test can be taken as a worst-case scenario which gives an upper bound of the performances of the protocol.

This evaluation, not only shows the performance of the topology service but also shows what performances we should expect from the other services. Indeed, the only step that will differ form service to service is specifically the computation part of the agreement phase. The data verification depends on the type of data, i.e. on the service. As shown in the last graph, this time is negligible with respect to other times.

VII Related Work

The lack of an efficient, transparent, distributed healthcare system for data sharing, and the problems of centralized databases encouraged researchers to find solutions based on the skipchain technology to achieve transparency, strong integrity, higher availability of data and lower costs (no intermediary). Designs to solve these problems have been proposed by others. We give here references to two projects that are the most similar to our:

- [10] proposes a solution based on a permissioned distributed ledger which tracks every change of data, records and controls access through smart contracts and digital identities. Moreover, on top of the skipchain, it implements a layer for secure and privacy preserving processing, so that only permissioned identities can access confidential information. Data must remain encrypted during storage and processing in personal data stores.
- [3] aims to empower the healthcare ecosystem with the skipchain technology without substituting the existing technology. The storage of medical information on traditional databases requires techniques to tie that data with the skipchain, enabling only authorized users to access it, e.g data can be accessed via a secure hash function stored in the skipchain, such as through encrypted pointers and hash of pointed data.

However, the modularity of our solution has important advantages on the management of the stored information and the code. Moreover, our services target a broader variety of data (e.g. topology, user identity, etc.) and are therefore employable in a more general-case scenario, whose main goal is guaranteeing accountability. Finally, the permissioned distributed ledger implementation employed in our project leads to high scalability and low transaction latency.

VIII Conclusion

We have proposed a protocol which enables distributed accountability while also preserving transparency and strong integrity. One of the main features is that the services are extremely ductile, in the sense that they can be adapted and employed in other environments under the initial few assumptions described in the threat model. Other ad-hoc services can be implemented on top of the skipchain service to complement those we propose in this project.

Regarding the limitation of the services, our trust model assumes that the institutions provide correct data. However, they could grant malicious users access to sensitive data by assigning them to a group with high privileges. This is a potential risk, therefore, the actions of the institutions must be monitored in case they are not trusted. In that sense, accountability property that we enabled can mitigate this issue by distributedly recording all operations, even the malicious ones. Accountability does not avoid the problem but allows the system to detect it and react to a dangerous situation.

At the moment of writing this paper, the major structural parts of the four services are implemented, but some functionalities have to be completed. Apart from the Topology service, whose implementation and analysis has been performed, it remains to add and test the CoSi signature in the other services, in order to correctly sign and verify the data. After that, the next steps consist in deploying and evaluating all the services in a real environment (i.e. with the UnLynx framework). In particular, for the Query Logging service, a protocol to manage Merkle trees and queries has to be implemented. The evaluation is a key point: some possible changes could be necessary depending on the outcomes of the performance analysis and profiling of the four services. We foresee the efficiency of the system can be improved and that the system can be made scalable to a national environment.

References

[1] Mininet: An instant virtual network on your laptop (or other pc). `http://mininet.org/`.

[2] Use of blockchain in health it and health-related research. `http://www.cccinnovationcenter.com/challenges/block-chain-challenge/view-winners/`.

[3] Brodersen, C., Kalis, B., Leong, C., Mitchell, E., Pupo, E., and Truscott, A. Blockchain: Securing a new health interoperability experience. `https://www.healthit.gov/sites/default/files/2-49-accenture_onc_blockchain_challenge_response_august8_final.pdf`, 2016.

[4] EPFL-Dedis. BFTCoSi. Under implementation.

[5] EPFL-Dedis. CoSi. `https://github.com/dedis/cothority/tree/master/cosi`.

[6] EPFL-Dedis. Identity. https://github.com/dedis/cothority/tree/master/ident.

[7] EPFL-Dedis. Onet v1.0. https://github.com/dedis/onet/tree/v1.0.

[8] EPFL-Dedis. Skipchain_835_2. https://github.com/dedis/cothority/tree/skipchain_835_2/skipchain.

[9] Froelicher, D., Egger, P., Sousa, J. S., Raisaro, J. L., Huang, Z., Mouchet, C., Ford, B., and Hubaux, J.-P. Unlynx: A decentralized system for privacy-conscious data sharing. *Under Submission* (2017).

[10] Shrier, A., Chang, A., Diakun-thibault, N., Forni, L., Landa, F., Mayo, J., and van Riezen, R. Blockchain and health it: Algorithms, privacy, and data. https://www.healthit.gov/sites/default/files/1-78-blockchainandhealthitalgorithmsprivacydata_whitepaper.pdf, August 8, 2016.