Proposal of a trusted and transparent version for DP-3T : Trusted-DP3T

A. Adinolfi[1], R. Molinari[1], S. Radio[1], G. Seccia[1], and L. Sicuranza[1]

[1]*Department of Computer Engineering, Electrical Engineering and Applied Mathematics (DIEM)*
Email: {a.adinolfi40, r.molinari5, s.radio, g.seccia2, l.sicuranza1}@studenti.unisa.it

Abstract

Due to the spread of the SARS-CoV-2 virus, an increasing number of researchers and private companies are trying to find out a solution for tracking people contacts in order to prevent and monitor the virus spread. Contact-Tracing consists in a process to identify people who may have been in contact with an infected person and the possible subsequent collection of further information on these contacts. Most of the proposed systems are based on a core technology available to everyone and used in different scenarios, namely the "Bluethooth Low Energy" (BLE) technology already present in our smartphones. Although the mere contact tracing is itself a considerable challenge, the crucial point that differentiates the various solution proposals is the way in which the different techniques can maintain the privacy and secrecy of user data. Among the most important proposals, without going into details related to its birth, we report the one born from the collaboration of cryptographers and cyber security experts from all over Europe. The project is called Decentralized Privacy-Preserving Proximity Tracing (DP-3T). In this work, starting from the given protocol, the goal is to add a further level to guarantee the transparency of this system in order to optimally reflect the needs of such an application. The term "guarantee" means a set of operations and additional resources to the DP-3T system aimed to verify the reliability and transparency of the operations. This requirement will be achieved exploiting `notarization`, that is one of the possible use cases of distributed ledger technologies (DLT) and in particular of Blockchain technology, which constitutes the implementation with the greatest resonance and impact. *Keywords:* Cybersecurity, Blockchain, Notarization, DP-3T, Contact-Tracing, Timestamping.

Contents

1 Project proposal

- **Members**:
  - Adinolfi Alessandro 0622701075
  - Molinari Raffaele 0622701093
  - Radio Salvatore 0622701217
  - Seccia Giuseppe 0622700831
  - Sicuranza Lorenzo 0622701207
- **Project title**: Trusted DP3T
- **Project topic:** Ensuring transparency of a people contact-tracing system using pseudonyms on their own smartphones.
- **Involved entities:**
  - Uninfected citizen: He is a citizen to whom is not arrived any message communicating of being infected.
  - Infected citizen: He is a citizen to whom is arrived a message communicating he is infected.
  - Verifier: Any citizen or entity that wants to verify the consistency of the information stored on the server.
  - Analysis laboratory: It is the place to where a citizen can make a test and verify if he is positive to the virus or not.
  - Government: It handle a server to coordinate the information collected by the smartphones, to let verifiers validate its consistency. It also coordinates the laboratories to help patients to communicate their positiveness to the system.
- **Project goals:** The aim is to stimulate the app usage increasing the transparency between the government and the citizens. Particularly, it is used notarization on public Blockchain to make the server transactions public and verifiable. In this way, a verifier entity could find possible irregularities by who handle the server.
- **Expected tasks:**
  1. Description of a notarization protocol of the server state on Bitcoin Blockchain
  2. Description of the validation mechanism which allows to obtain transparency
  3. Integration of what explained in points 1) and 2) with the DP-3T protocol which ensures the privacy-preserving contact tracing.
  4. Implementations of what said in points 1) and 2)

2 Reliability and integrity properties issues

Among the various protocols created following the spread of COVID-19, the one considered is the Decentralized Privacy-Preserving Proximity Tracing (DP-3T). The goal of the proximity tracing system is to determine who has been in close physical proximity to a COVID-19 positive person and thus exposed to the virus, without revealing the contact's identity or where the contact occurred.

2.1 Description of the properties

The system based on DP-3T provides the following security and privacy properties:

- **Ensures data minimization** : the central server only observes anonymous identifiers of COVID-19 positive users without any proximity information. Health authorities learn no information except the ones provided when a user reaches out to them after being notified.
- **Prevents abuse of data** : as the central server receives the minimum amount of information tailored to its requirements, it can neither misuse the collected data for other purposes, nor it can be coerced or subpoenaed to make other data available.
- **Prevents tracking of users** : no entity can track users which have not reported a positive diagnosis. Depending on the implementation chosen, others can only track COVID-19 positive users in a small geographical region limited by their capability to deploy infrastructure that can receive broadcasted Bluetooth beacons.[1]

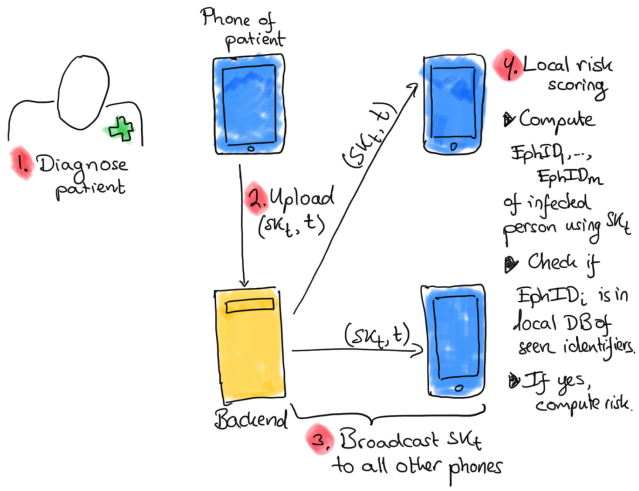

Figure 1. Proximity tracing process

Starting from the DP-3T protocol, as the centralization of the server on the backend leads to problems of trust with respect to data management, it is necessary to add a further level guaranteeing the following property:

- **Transparency** : Anyone should be able to check and ensure that the data stored and the operations performed by the server are reliable, consistent and verifiable. Malicious actions performed by the server or hypothetical adversaries need to be revealed by any system user.
- **Non-repudiation** : It refers to the ability of ensuring that a party to a contract or a communication cannot deny the authenticity of its signature on a document or the sending of a message which it has originated. In the context of DP-3T protocol, server must be fair with the respect of users who send keys to the system.

2.2 Description of adversarial scenarios

Additional properties proposed in this project, as mentioned above, arise from the possible scenarios in which DP-3T server management could operate maliciously.
With respect of **transparency** and **non-repudiation** properties, the objectives are to reveal any possible operation that may violate them without knowing if those operations were generated by a malicious system management or a slanderer client.

1. It is possible that could exist an interest, by those who manage the server, to remove or modify one or more `(Sk,t)` of people declared infected immediately after key insertion or at a generic time during the system life-cycle. For example, suppose that there is an influent person who changes his idea or decides that his key or a friend's key has to be removed from the system for a certain reason. A system that is able to perform such operation violates *transparency* and surely can not be trusted from other users. Therefore it is necessary to implement an addition to DP-3T protocol to avoid such a possibility.

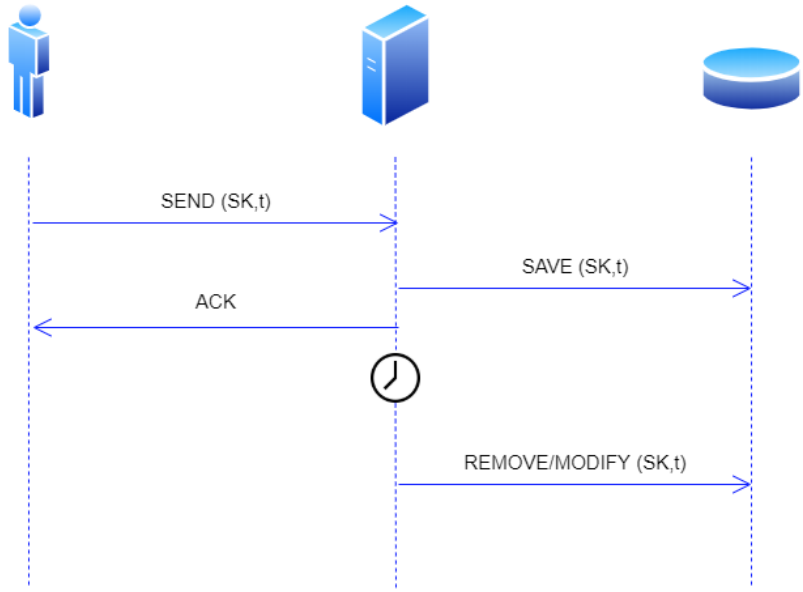

Figure 2. Adversarial scenario for transparency.

2. Server management could act in such a way to exclude from the system a particular user or a group of them so that contact tracing is not efficient. As an example, consider a despicable leaders of opposition political movements who may want to let partially spread the epidemic in order to attack the government and corrupt system administrators. It is clear that *Non-repudiation* of operations is a crucial feature of the system. On the other hand, another scenario could be the opposite situation in which a malicious user would accuse the server to have denied his key and server must provide evidence of the contrary.

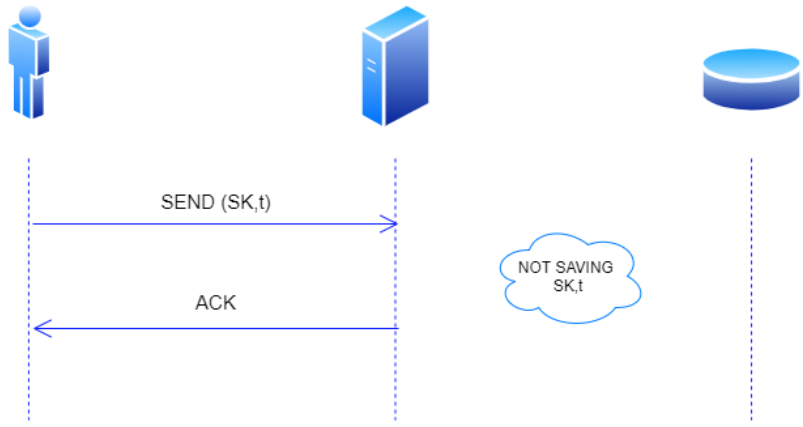

Figure 3. Adversarial scenario for non-repudiation.

Referring to provided scenarios, it is possible to note how the server administrator has been considered as the system adversary. Actually, there could exist other external adversaries, which could realize the mentioned malicious behaviours just to modify the data integrity or to discredit the server. From this point of view, an adversary could be identified as any kind of entity who can break the server access security. For instance, it could modify the data stored in the database using *sql-injection*. From this point of view, adversaries could attack any kind of service and not just the DP-3T ones. Consequently, it would be considered only the aspects referred to the reliability of the server and its operations.

3 Trusted-DP3T : system upgrade proposal

To achieve the objectives described above, it has been formulated a solution based on **timestamping** and **notarization** on Blockchain. In particular, it has been considered **Bitcoin**, which is the most popular public Blockchain.
Timestamping technique is used for reaching both the purposes but in two different approaches that will be explained in the following sections. First approach uses standard timestamping protocol as described in IETF-RFC3161 [6] in order to obtain non-repudiation from the server, while the second applies notarization on public Blockchain, which uses as main component timestamping, in order to achieve transparency over time.
In the next sections, details about notarization and timestamping will be discussed before explaining proposed system architecture and protocols of the proposed solution.
From now on with the term *notarization* is intended to refer to the act of "timestamping on public Blockchain", *timestamping* is referred to the protocol described by RFC3161 and *timestamp* is the proof generated by the protocol implied by the context.

3.1 Timestamping and Notarization

3.1.1 Timestamping

Timestamping has become a component function of the fraud-deterrent process of notarization, which is used to assure the authenticity of documents, records and other assets. A timestamp can be seen as a sequence of characters representing the time in which an event occurred. A proper definition of a timestamp could be given as:

> **Def.** *A **timestamp** is a proof that some data **d** existed prior to a certain time **t**.*

A Time Stamping Autority (TSA) may be operated as a Trusted Third Party (TTP) service and must provide "proof-of-existence" of a particular datum at a given time. This protocol may be used as a building block to support non-repudiation services. For example, to verify that a digital signature was applied to a message before the corresponding certificate was revoked. This is an important public key infrastructure operation. The TSA can also be used to indicate the time of submission when a deadline is critical, or to indicate the time of transaction for entries in a log.
Time-Stamp Protocol (TSP) is based on three kind of operations namely *Request*, *Response* and *Verify*, which describe its behaviour. The aim of this section is not to explain the protocol, details of which can be found in the IETF-RFC3161. Notice that this protocol has direct implementation in the openSSL library and definition of usage over HTTP.

3.1.2 Notarization

> "Notarization is the official fraud-deterrent process that assures the parties of a transaction that a document is authentic, and can be trusted. It is a three-part process, performed by a Notary Public, that includes of vetting, certifying and record-keeping."
>
> National Notary Association

The notarization [4] is traditionally achieved by *certification authorities* (**CA**). However, a central authority represents a single point of failure because who maintains that ledger could act maliciously and easily tamper some data for his own interests. So we need to decentralize the source of trust and grant the reliability of that ledger. A solution could be easily achieved taking advantage of the Bitcoin Blockchain technology.

Notarization of digital documents is one of the most famous non-monetary application of the Bitcoin Blockchain. Blockchain technology assures the integrity of data once written in the chain. The promises like tampering resistance, non repudiation and its traceability make Blockchain a good candidate to provide some of the notarization capabilities.

Despite Blockchain is not able to fully replace the notarization service yet, it can provide almost all features to support that service. It has been already introduced *timestampig* which is able to produce proof called *timestamp* but this time the proof-of-existence of a document can be retrieved directly in the Bitcoin Blockchain instead of a private document holded by interested parties. Timestamp proof does not require to publish the entire document on the Blockchain, but only a commitment to it, that is something caused by the input *d*.
A good commitment operation is the *hash function*. In particular, it has been used SHA256 to implement it. The result of *hash function* will be stored inside the transaction using the **OP_RETURN** script. This is a special kind of transaction, called *null data transaction*, that will not be stored in the UTXO dataset. UTXO stands for the unspent output from bitcoin transactions. Each bitcoin transaction begins with coins used to balance the ledger. UTXOs are processed continuously and are responsible for beginning and ending each transaction. Confirmation of transaction results in the removal of spent coins from the UTXO database. But a record of the spent coins still exists on the ledger In this way it is possible to avoid to fill the dataset with unspendable data.
Summing up, Blockchain timestamping provides a public proof of existence of a digital document, that cannot be faked or removed and without the need of a central trusted authority. However, this procedure has a downside : lacking a proper standardization, although in 2012 work began on an open-source project , namely **OpenTimestamps**, that aims to provide a standard format for Blockchain timestamping, and efficiently solve this downside.

3.1.3 *OpenTimestamps*

OpenTimestamps defines a set of rules for conveniently creating provable timestamps and later independently verifying them without relying on a trusted third party; this represents an enhancement in term of security, since it excludes the possibility of a malicious (or careless) notary to compromise the timestamp.

Currently this protocol fully supports Bitcoin Blockchain timestamping. A timestamp proof made with OpenTimestamps consists in a list of commitment operations applied in sequence to the doc-

ument, ending with one or more time attestations. Such a list of operations is actually a tree of operations, with the document as the root, the commitment operations as the edges and time attestations as the leaves. Anyone can easily verify the proof just replaying the operations and checking that the final result is a message that you already know existed at a certain time. Since commitment operations grant that different inputs will always result in different outputs, we are sure that there is no way to change the original document without invalidating the proof.

In addition to this, OpenTimestamps comes with a system of calendar servers, where anyone can submit a digest to be timestamped. As its name suggests, a calendar server grants remote access to a calendar, a collection of timestamps. Notice that the calendar server learns nothing about the original documents, it only collects "meaningless" digests. Each file is protected by a nonce, thus files are not directly connected in any way. It makes the promise that every submitted digest will be timestamped by the Bitcoin Blockchain in a reasonable amount of time, and keeps indefinitely all completed timestamps, publicly available in every moment. Hence, the proof is generated in about one second. Obviously, the inconvenience here is that public calendar servers are a central point of failure, but this represents just a little inconvenience, calendar servers remain trustless in the sense that they cannot tamper a proof, because it is Bitcoin and not theirselves to provide the validity of a timestamp. In fact, a proof made with the aid of a calendar server is called incomplete.

In order to timestamp the transactions, the calendar spends bitcoins from its own wallet. To spend a fixed amount each day, it applies the replace-by-fee policy. In means that the calendar makes a transaction with the lowest possible fees, recursively increasing that amount by a parameter a until the transaction enters in a block. However, it could happen that some timestamps takes too long to be completed.

3.2 Trusted-DP-3T design

The current architecture of the DP-3T protocol provides the following server-side structure:

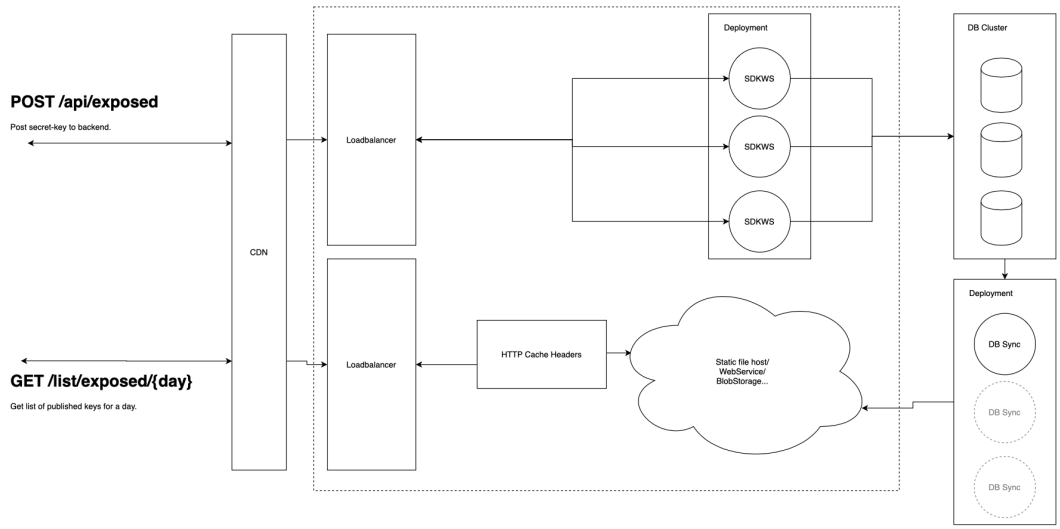

Figure 4. Backend of DP-3T

This server provides the necessary information for the SDK to initialize itself. After the SDK loads the base URL for its own backend, it will load the infected list from there, as well as post if a user

is infected. This will also allow apps to fetch lists from other backend systems participating in this scheme and can handle roaming of users [2].

The API is composed by:

- `POST /api/exposed` : used to load the (Sk,t) once the application has started.
- `GET /list/exposed/{day}` : used to receive the (Sk,t) of the infected of the day. We also pass on the `day` value we are looking for.

For the data to be stored, a database is used with the following table:

t_exposed	
PK	**pk_exposed_id: Serial**
	key: Text NOT NULL UNIQUE
	received_at: Timestamp with time zone Default now() NOT NULL
	key_date: Timestamp with time zone NOT NULL
	app_source: Character varying(50) NOT NULL

Figure 5. Database table

Where the attributes indicate:

- **pk_exposed_id** : It represents an incremental key to identify the item in the database
- **key** : Contains the `(Sk,t)` value
- **received_at** : Time value indicating when the `(Sk,t)` was received
- **key_date** : Contains the `t` value
- **app_source** : App source from which the data comes

3.2.1 Implementing notarization service on Bitcoin

Starting from the current server architecture, it is necessary to add some components to satisfy the properties discussed in the previous sections.
The first property that is going to be treated is *transparency*. From the previous section it can be easily understood that notarization is the key feature to satisfy it.
Therefore the main component to add to DP-3T architecture is a *OpenTimestamps* server merged with the DP-3T server. *OpenTimestamps* server alone is not sufficient because it must be implemented even a standard **notarization routine** in order to make DP-3T operations verifiable. Details about the setup of *OpenTimestamps* server could be found in the section 5.
With the term *notarization routine* is intended the set of operation to execute periodically. To notarize in publicly known format, the proposed period is one hour.

The operations composing the notarization routine, proposed at the end of each hour, are:

1. Getting the `(Sk,t)` received during the considered hour and format a document as:

```
Sk1,t1
Sk2,t2
Sk3,t3
...,...
```

2. Notarization of the document digest intended as the hash value of the formatted document. The notarization process is described in OpenTimestamps.
3. Storage of the `.ots` file that contains the proof-of-existence of the document in Bitcoin Blockchain, which could be requested by any verifier who would check the operation over the time.

The calendar server as described in OpenTimestamps protocol [5], after taking in care the notarization request, runs the `OP_RETURN` script in order to put the digest in a transaction. While the transaction is not confirmed and added in a block of the Bitcoin Blockchain, the proof produced is considered "incomplete". Incomplete timestamps are ones that require the assistance of a remote calendar to verify. When the transaction is validated, the calendar provides the path to the Bitcoin block header upgrading the timestamp that can be verified without the usage of the calendar server.

The verifier can be any client node that has:

- the document that contains the list of the (Sk,t) stored into the DP-3T server in a certain day `y` and at a certain hour `x`
- the complete proof of existence

The verifier has to take the digest of the document containing the list of the considered (Sk,t) and repeat the set of commitment operations indicated in the proof; the result that it obtains after this operations must be compared with the `OP_RETURN` value of the transaction, whose ID is indicated in the proof. If the comparison fails, it means that someone has tampered with the integrity of the data. The procedure described is shown in the following scheme:

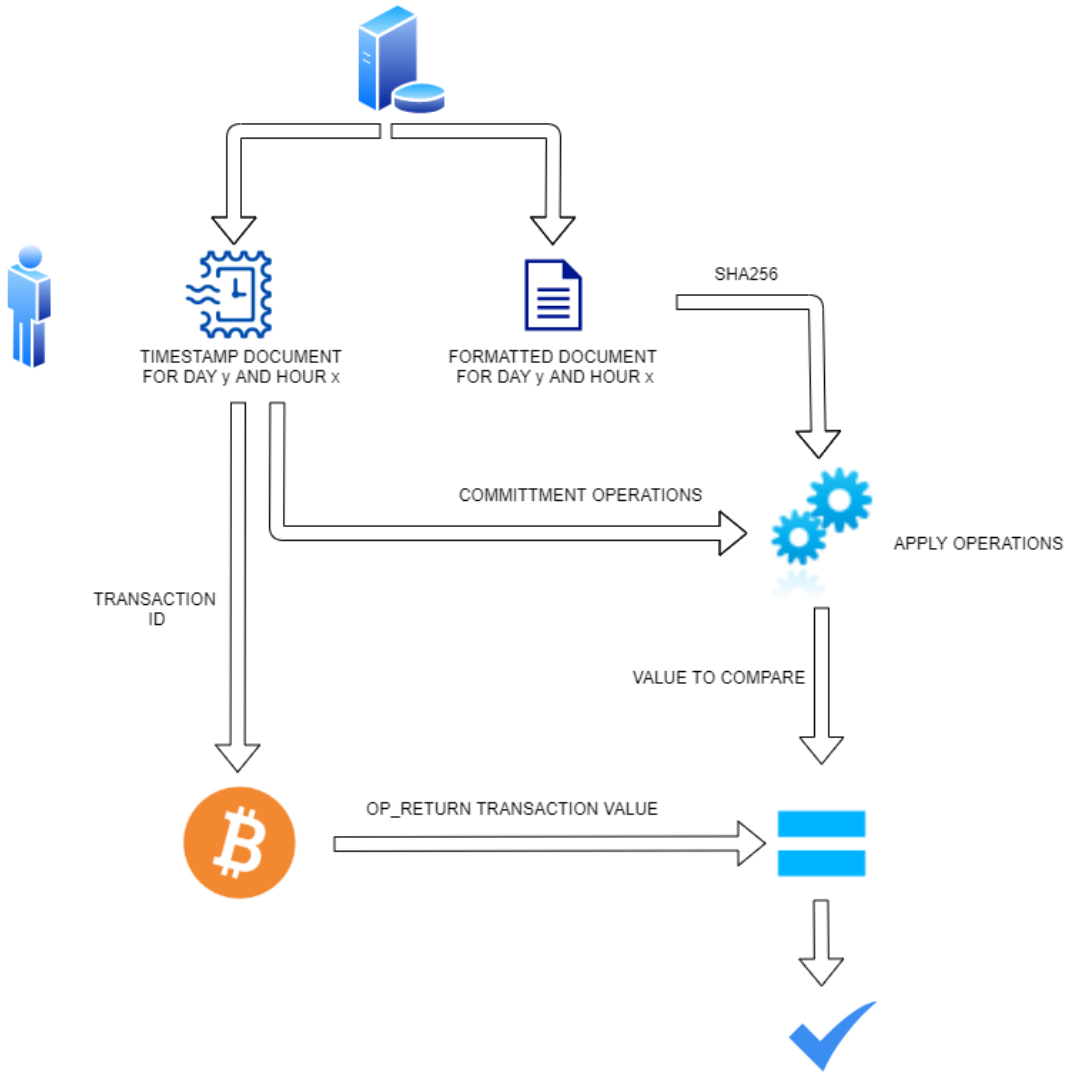

Figure 6. Backend of Trusted-DP3T

More details about the verification procedure are provided in section 5.
In this new architecture, two new API calls must be provided in addition to existent ones:

- `GET /proof/exposed/{day}/{hour}` : used to receive the proof of existence of the document containing the list of the (Sk,t) stored into the DP-3T server in a certain day y and at a certain hour x.
- `GET /list/exposed/{day}/{hour}` : used to receive the list of the (Sk,t)s stored into the DP-3T server in a certain day y and at a certain hour x.

The final structure is the following:

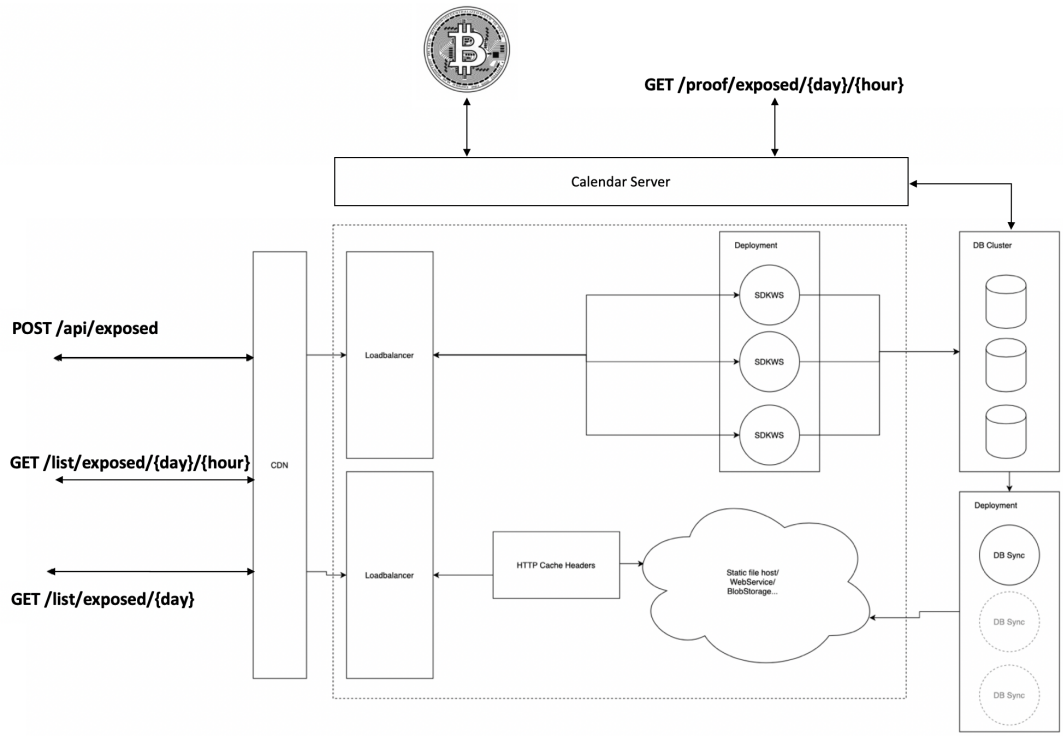

Figure 7. Backend of Trusted-DP3T

The choice to use a one hour-time window for the notarization procedure is given by the will to limit as much as possible eventual malicious adversary's behaviours which could modify or delete the `(Sk,t)` sent by the client. In this way, each hour, all the data arrived to the server will be stored on the Blockchain and will become immutable.

3.2.2 Implementing non-repudiation scheme

It has been depicted implementation details about how it is possible to ensure transparency over time of server operations. The next step requires to satisfy the *non-repudiation* property, in order to be sure that if server receives a couple `(Sk,t)`, it is not able to deny to have received it.
To achieve the requirement, a non-repudiation protocol scheme is provided. This protocol supposes the existence of a TTP named as `Non-Repudiation Server` (NRS) which is responsible to manage conflict situations.
The notation used in this paragraph is the following:

- `r` : the distinguishing identifier of the originator randomly choosen.
- `SERVER_ADDRESS`: the distinguishing identifier of the recipient.
- `(Sk,t)` : the message to be exchanged.
- `h(M)` : a one-way hashing function applied to message M, SHA256 has been considered.
- `[M]dsgX` : the digital signature of entity X appended to the message M.
- `ts1, ts2` : the time attestations.

- `{ M1,M2 }` : the concatenation of messages M1 and M2 without any change to their contents.
- `S` → `R : {M}` : This means that the entity S sends message M to entity R.

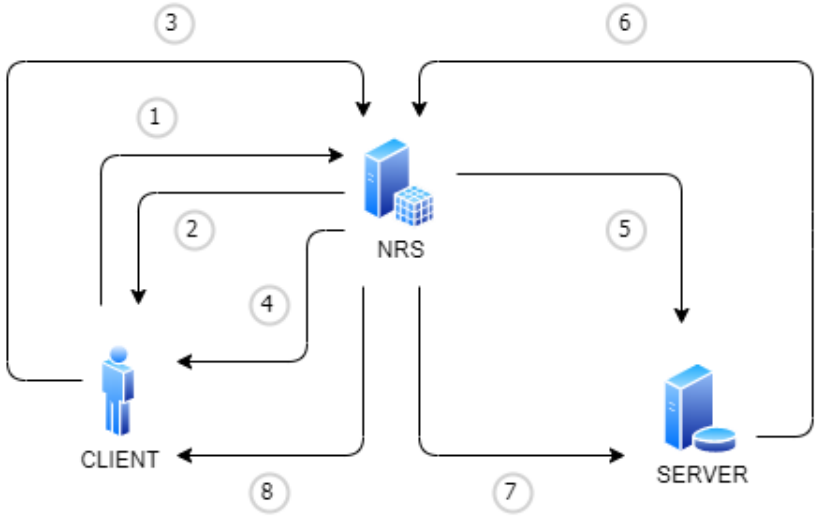

Figure 8. Non-repudiation scheme

Note that each communication has been supposed secure in the sense that the communication is encrypted using TLS protocol. Steps of non-repudiation scheme in figure are:

1. client → NRS : {r}
   Before entering into any communication with the NRS, client must first generate a random number identifier for handling disputes if necessary.

2. NRS → client : {[r]dsgNRS}
   The NRS sends a confirmation to the client that from now on its communication will be referred as r.

3. client → NRS : {p_poo}
   Client generate a partial proof of origin (p_poo) which includes message to send formatted as {(Sk,t)}. p_poo is the following:
   p_poo = [r, SERVER_ADDRESS, {(Sk,t)}]

4. NRS → client : {POO}
   The NRS adds a time-stamp to the received p_poo to form a complete POO (Proof Of Origin).
   POO = [p_poo, NRS, ts1]dsgNRS
   The POO is sent to client who can verify that the time-stamp is valid.

5. NRS → server : [r, p_por]dsgNRS
   The NRS initiates the proof of receipt generation by forwarding the p_por (partial proof of receipt) along with the r identifier to the recipient server.
   Note that p_por does not includes the message but only a commitment to it.
   p_por = SERVER_ADDRESS, r, h(POO)

6. server → NRS : s_p_por
   Server signs the received p_por with his secret key to form the signed partial proof of receipt s_p_por as follows:
   s_p_por = [SERVER_ADDRESS, r, h(POO)]dsgB

7. NRS → server : {POO}
   The NRS stores a copy of the POR and POO for later use. The POO is sent to the recipient who can examine the data contained therein. Note that POR (Proof Of Receipt) is the timestamped version oh s_p_por as:
   POR = [s_p_por, NRS, ts2]dsgNRS

8. NRS → client : {POR}
   The NRS sends the POR to the client.

If a dispute arises in relation to a non-repudiable data exchange, an agreed arbitrator which is trusted by both principals may be called upon to resolve it. The entities involved in the exchange present evidence to this arbitrator who makes use of arbitration rules to reach a decision regarding the exchange. Three entities may submit evidence - the NRS, the originator and the recipient. The arbitrator's decision-making rules must allow for the submission of any combination of evidence by these three entities. In reaching a decision, the arbitrator will deem that a data exchange did not take place in the absence of sufficient evidence to prove otherwise. The decision-making rules are as follows:

- If client knows POR, then this is proof that the server received the data.
- If recipient knows POO, then this is proof that the client did send the data which the recipient now possesses
- If NRS knows POR, then this is proof that the protocol at least successfully completed step 6, by which stage, the data exchange is deemed to have taken place.
- If NRS knows POO, then this is proof that the protocol at least successfully completed step 3, by which stage, the data exchange is not deemed to have taken place.

Using these four rules, it is possible to reach a decision regarding any claimed non-repudiable exchange. For instance, if the client submits a POR, then regardless of whether any of the other entities submit evidence, the arbitrator knows that the exchange took place. Table 1 shows the different possibilities of evidence submission and the decisions which follow from them. The symbol 'X' indicates a "don't care" state.

Decision table in dispute

NRS submits		Client submits	Server submits	Data
POO	POR	POR	POO	Exchange
X	YES	X	X	YES
X	X	YES	X	YES
X	X	X	YES	YES
X	NO	NO	NO	NO

Table 1. Handling disputes with non-repudiation scheme

4 Analysis of confidentiality and integrity of the proposed system

Recalling what was described in section 2, in this section an analysis of the proposed properties and solutions, with respect to the adversaries presented in considered adversarial scenarios, is shown.

4.1 Transparency adversarial scenario

It is recalled that the adversary *Adv* in this scenario is the server who is interested in modifying or removing one or more keys from its own database for a certain reason. In order to identify these kind of operations, notarization into the DP-3T base system has been introduced. Given a document *doc*, containing all the *(Sk,t)* sent on day *d* at hour *h*, which is stored in the `OP_RETURN` field of the correspondent Blockchain transaction.
It is supposed that the adversary *Adv* modifies or removes one or more *(Sk,t)* to compromise the data integrity. It will be denoted as *doc'* the document containing the *(Sk,t)* after the modifications. With the introduction of the notarization mechanism, any third party entity, such as a simple citizen or an external authority, could always verify if what is stored into the database corresponds to what is stored into Blockchain following comparison steps seen in Figure 6. If they don't corresponds it will be obvious that the data integrity has been compromised and a system attack is detected.

4.2 Non-repudiation adversarial scenario - malicious server

In this scenario the adversary *Adv* is the server but the type of the attack changes. In this case, the adversary is interested in ignoring one ore more incoming *(Sk,t)* before the beginning of the notarization procedure.
Suppose that during a certain hour *h*, a client *A* sends his own *(Sk,t)* to the server *S* through the trusted entity *NRS*. So *A* begins the protocol described in the 3.2.2 section, at the end of which the client *A* will receive the *POR* certifying that his *(Sk,t)* has correctly reached the server *S* at the time *ts2*. If the server *S* will not insert the key *(Sk,t)* inside the database, the client *A* could exhibit his *POR*, that has been signed by the server S and validated by the *NRS*. For this reason, the client has the possibility to show the correct sending of the key. Thus, the data non-repudiation property has been respected.
It is necessary to note that in case of disputes, it will be needed that the client should expose his own *(Sk,t)* to demonstrate that the *POR* is valid for the key under examination and that this key is not present inside the DP-3T database.
However, considering that in this case the reliability of the server has been compromised, it is legit to think that the server will be discouraged to act in this way.

4.3 Non-repudiation adversarial scenario - malicious client

This time, it is supposed that the server will respect the non-repudiation property. In this attack, the adversary *Adv* is the client *A*, whose objective is to discredit the server reliability.
The client wants to deny the presence of his own key on the server database despite the fact that he has sent it.

Using the provided protocol, to carry out the attack, the client should provide the *POO* to verify the absence of the *POR* in the *NRS*.
However, the server could still avoid his accusation maintaining the POO of the analyzed *(Sk,t)*. The condition in which the server or the *NRS* demonstrates the inconsistency of the client accusation is associated to a "false alarm". The key *(Sk,t)* will be exposed without affecting the server reliability. For this reason, the client is discouraged to put himself in such a situation and he should give the alarm only if he can correctly verify the absence of his key of the document sent on the server and notarized on the Blockchain.

5 Implementing notarization with *OpenTimestamp*

As previously mentioned, it has been implemented Blockchain notarization using OpenTimeStamps and the Bitcoin Blockchain. Before being able to make a notarization, it's required an installation and configuration phase of:

- an OpenTimeStamps client
- an OpenTimeStamps server
- a Bitcoin Blockchain in regression test mode

It is necessary to open two different terminals:

- the first one to set up the Bitcoin Blockchain in regression test mode and the Opentimestamps server
- the second one to set up the Opentimestamps client and use it

5.1 Setting-up Bitcoin Blockchain in regression test mode

In order to test the solution proposed with the OpenTimestamps Calendar server running in local environment it is needed as requirement to get a full node of Bitcoin Blockchain running on the server. Since both Mainnet and Testnet nodes of Bitcoin takes too much space on disk and for transaction validation we have to wait the work of actual miners it has been chosen to run a regression test Bitcoin chain in order to take off all testing disadvantages.

Therefore, we will setup a regression test Bitcoin chain as a playground where we can observe how things work and how to interact with the Blockchain. As this chain has no value and it is possible to generate new blocks whenever is necessary, it is ideal for learning and testing without actually risking anything. In the next paragraph is shown a simplified version of installation tutorial proposed in *"Learning Bitcoin from the Command Line"* [3] and is supposed to run in Linux environment under "trusteddp3t" username.

5.1.1 Download and verify Bitcoin core files

First of all is needed to download correct version of bitcoin-core. Notice that OpenTimastamp server require version 0.17.1:

```
$ export BITCOIN=bitcoin-core-0.17.1
$ export BITCOINPLAIN=`echo $BITCOIN | sed 's/bitcoin-core/bitcoin/'`
```

Now grab relevant files from bitcoin:

```
$ wget https://bitcoin.org/bin/$BITCOIN/$BITCOINPLAIN-x86_64-linux
-gnu.tar.gz -O ~trusteddp3t/$BITCOINPLAIN-x86_64-linux-gnu.tar.gz
$ wget https://bitcoin.org/bin/$BITCOIN/SHA256SUMS.asc
-O ~trusteddp3t/SHA256SUMS.asc
$ wget https://bitcoin.org/laanwj-releases.asc
-O ~trusteddp3t/laanwj-releases.asc
```

Look at the signature to verify files:

```
$ /usr/bin/gpg --import ~trusteddp3t/laanwj-releases.asc
$ /usr/bin/gpg --verify ~trusteddp3t/SHA256SUMS.asc
```

Among the info you get back from the last command should be a line telling that you have a "Good signature". If warning was raised it could be ignored for this setup.
Next, you should verify the Hash for the Bitcoin tar file against the expected Hash:

```
$ /usr/bin/sha256sum ~trusteddp3t/$BITCOINPLAIN-x86_64-linux-gnu.tar.gz |
awk '{print $1}'
$ cat ~trusteddp3t/SHA256SUMS.asc |
grep $BITCOINPLAIN-x86_64-linux-gnu.tar.gz |
awk '{print $1}'
```

If those both produce the same number everything is fine.

5.1.2 Installing Bitcoin

At this point in order to install Bitcoin few commands are needed:

```
$ /bin/tar xzf ~trusteddp3t/$BITCOINPLAIN-x86_64-linux-gnu.tar.gz -C ~trusteddp3t
$ sudo /usr/bin/install -m 0755 -o root -g root -t
/usr/local/bin ~trusteddp3t/$BITCOINPLAIN/bin/*
$ /bin/rm -rf ~trusteddp3t/$BITCOINPLAIN/
```

5.1.3 Create Bitcoin Configuration

Last step is to set up a bitcoin configuration file. The file proposed is appropriate for an unpruned testnet setup, except for regtest=1 and testnet=0 assertion that should be inverted if regtest is not necessary, but since the core will run in regression test mode the only difference is that we do not need to wait untill all block are downloaded from testnet, which is at time of writing about 20GB.

First step is to create the configuration directory:

```
$ /bin/mkdir ~trusteddp3t/.bitcoin
```

And put the configuration file proposed:

```
$ cat >> ~trusteddp3t/.bitcoin/bitcoin.conf << EOF
# Accept command line and JSON-RPC commands
server=1

# Set database cache size in megabytes (4 to 16384, default: 450)
dbcache=1536

# Set the number of script verification threads
#(-6 to 16, 0 = auto, <0 = leave that many cores free, default: 0)
par=1

# Set to blocksonly mode,
# sends and receives no lose transactions,
# instead handles only complete blocks
blocksonly=1

# Tries to keep outbound traffic under the given target
# (in MiB per 24h), 0 = no limit (default: 0)
maxuploadtarget=137

# Maintain at most <n> connections to peers (default: 125)
maxconnections=16

# Username for JSON-RPC connections
rpcuser=bitcoinrpc

# Password for JSON-RPC connections
rpcpassword=$(xxd -l 16 -p /dev/urandom)

# Allow JSON-RPC connections from, by default only localhost are allowed
rpcallowip=127.0.0.1

# Use the test chain (0 if using regtest)
testnet=0
# Delete if not using regtest
regtest=1
# settings to run OpenTimestamps server (delete if not using it)
deprecatedrpc=signrawtransaction
deprecatedrpc=accounts
addresstype=bech32

# Maintain a full transaction index,
# used by the getrawtransaction rpc call (default: 0)
txindex=1

# Make the wallet broadcast transactions (default: 1)
walletbroadcast=1
EOF
```

Note that the following fields:

```
deprecatedrpc=signrawtransaction
deprecatedrpc=accounts
addresstype=bech32
```

are set that way in the configuration file to allow the use of the Opentimestamps server.

To end, limit permissions to your configuration file:

```
$ /bin/chmod 600 ~trusteddp3t/.bitcoin/bitcoin.conf
```

5.1.4 Running Bitcoin in regression test mode

To start Bitcoind (Bitcoin Daemon) on Regtest and create a private Blockchain, you have to use the following command:

```
$ bitcoind -regtest -daemon
```

From now on user is able to run bitcoin-cli commands useful to the aim of project and shown in the previous sections.

5.1.5 Resetting the Regtest Blockchain

Regtest wallets and block chain state (chainstate) are saved in the regtest subdirectory of the Bitcoin configuration directory:

```
trusteddp3t@mybtc:~/.bitcoin# ls
bitcoin.conf  regtest  testnet3
```

If you want to start a brand new Blockchain using regtest, all you have to do is delete the regtest folder and restart the Bitcoind:

```
$ rm -rf regtest
```

5.2 Setting-up OpenTimestamps Calendar Server

This section sintetize instructions provided at *https://github.com/opentimestamps/opentimestamps-server*. Notice that is needed a local Bitcoin node version 0.17.1 with a wallet with some funds in it in order to run the server; a pruned node or regression test Blockchain is fine. While otsd is running the wallet should not be used for other purposes, as currently the Bitcoin timestamping functionality assumes that it has exclusive use of the wallet.

5.2.1 Install OpenTimestamps Server

First step is to clone or install repository code into the server and install requirements:

```
$ git clone https://github.com/opentimestamps/opentimestamps-server.git
$ cd opentimestamps-server
$ pip3 install -r requirements.txt
```

5.2.2 Create the calendar server

```
$ mkdir -p ~/.otsd/calendar/
$ echo "http://127.0.0.1:14788" > ~/.otsd/calendar/uri
$ dd if=/dev/random of=~/.otsd/calendar/hmac-key bs=32 count=1
```

The URI determines what is put into the URI field of pending attestations returned by this calendar server.
The HMAC key should be kept secret; it's meant to allow for last-ditch calendar recovery from untrusted sources, although only part of the functionality is implemented. More details about this feature are not provided in this document.

5.2.3 Run the server using Bitcoin *regtest* net

To actually run the server, run the otsd program. Proper daemonization isn't implemented yet, so otsd runs in the foreground. To run in testnet or regtest, use the –btc-testnet or –btc-regtest options. The OpenTimestamps protocol does not distinguish between mainnet, testnet, and regtest. For our purpose running Calendar Server in regtest mode is sufficient.

```
$ ./otsd --btc-regtest
```

5.3 Setting-up OpenTimestamps client

OpenTimestamps client is a command-line tool to create and validate timestamp proofs, with the Bitcoin Blockchain as a notary.

Before installing the OpenTimestamps client, it must be noticed that while you can create timestamps without a local Bitcoin core node, to verify proofs you need one (a pruned node is fine too). Standing the above consideration, the increased difficulty in using a command-line tool rather than a web interface is rewarded by a complete decentralization, you do not have to rely on any third part.

5.3.1 Install OpenTimestamps client and all dependencies

Install client (Python3 required):

```
$ pip3 install opentimestamps-client
```

Install the necessary dependencies:

```
$ sudo apt-get install python3 python3-dev python3-pip
python3-setuptools python3-wheel
```

Move to the directory that contains the file to be timestamped:

```
$ sudo mv [PATH TO THE FOLDER]
```

5.3.2 Requesting the timestamp of the file from the server

Since calendar server requires a local bitcoin node with at least some funds in it, before requesting timestamping of the file, at least 101 blocks must be generated to add Bitcoin in the wallet:

```
$ bitcoin-cli -regtest generate 101
```

To use your calendar server, tell your OpenTimestamps client to connect to it. In our application, since the DP-3T backend and the calendar server are run on the same machine it will not be necessary to establish a secure connection via the TLS protocol:

```
$ ots --btc-regtest stamp -c http://127.0.0.1:14788 -m 1 FILE
```

At this time the client must have automatically downloaded in the current directory a file with a .ots extension, named filename.txt.ots that is the proof of its timestamp.

5.3.3 Verify and upgrade the proof

Verify the proof:

```
$ ots --btc-regtest verify filename.txt.ots
```

Notice that you can't verify immediately the aforementioned proof. In fact, initally the proof is incomplete, so a verifier has to ask the remote calendars for the rest of the proof. It takes a few hours for the timestamp to get confirmed by the Bitcoin Blockchain (generally 6 confirmations). It is suggested to use regression test node to nullify this time.

Check for the current status of your proof file with the info command and get detailed proof information:

```
$ ots --btc-regtest info filename.txt.ots
```

Incomplete timestamps can be upgraded using the upgrade command which adds the path to the relative block header merkleroot to the proof. Upgrading the proof isn't available until the calendar server doesn't complete the attestation.

Before requesting the upgrade of the proof to obtain the complete proof, indicating that the timestamping of the file has been completed, it is necessary that other blocks are inserted within the Blockchain, using the `generate` command of the bitcoin command line interface previously seen if using regtest chain as suggested:

```
$ bitcoin-cli -regtest generate [number of the blocks to generate]
```

until the calendar server, that runs in another window of the terminal in foreground, indicates that your notarization request has been confirmed as can be seen below:

```
tx d59099321d585e82f663fec2e06a5d0381a1056e11850c76dd4fc6727d4f32a2
 fully confirmed, 1 timestamps added to calendar
```

Figure 9. Timestamp confirmation output by calendar server

Upgrade the proof taking into account that OpenTimestamps clients have a whitelist of calendars they'll connect to automatically; you'll need to manually add your new server to that whitelist to use it when upgrading or verifying:

```
$ ots --btc-regtest -l http://127.0.0.1:14788 upgrade FILE.ots
```

Once timestamping has been successful, you only left to use the verify command another time to check which block attested your timestamp (add the flag -v to be verbose).

```
$ ots --btc-regtest -v verify filename.txt.ots
```

It is important to make sure to tell your client what chain to use when verifying if running in testnet or regtest mode.

5.3.4 How to manually verify the integrity of the document

After the notarization of a document, it is possible to execute some operations to verify its integrity. These operations are stored inside the *.ots* file and could be visualized using the `info` command of the ots-client already seen. The result of this command is shown in the following picture:

```
$ ots --btc-regtest info filename.txt.ots
```

Figure 10. Example of info command

```
File sha256 hash: d1bdcd7ea02d68761710c5f52fc6f0156aca1b96093ec7f6e1f61478ebe1e70f
Timestamp:
append 8959f05892eed6330359086254cc0c23
sha256
append 825c1527b8539f5f81155f0f05722a7e
sha256
prepend 5ed7cb22
append e896695e7a3ea7ae
verify PendingAttestation('http://127.0.0.1:14788')
sha256
prepend 0100000001c774da5879b4446fe913d30f9f5b9b1a58015742fa98f83a54c8126aaa74d1940
000000048473044022041 29dc24eef9a62f0cffd8256042b976ecd2ea048edd830d51ae6f2c84457dfd
022015c219565d4d9c35c2b875fb9f031e59467638f2ab06d8955ee99773f5cf1fef01fdffffff0294f
0052a0100000016001404104f9d02fe172c197fe5c4de9183978d9666c30000000000000000226a20
append 67000000
# Transaction id d59099321d585e82f663fec2e06a5d0381a1056e11850c76dd4fc6727d4f32a2
sha256
sha256
prepend 9ae8ac5fba42c847a1d66672e95740254ea6fe2e279eb227da4ae9ca9299c23d
sha256
sha256
verify BitcoinBlockHeaderAttestation(104)
# Bitcoin block merkle root 25cac305a5fc7c9b3a0499880014e9a306bfb66f5b3d6150d94d06a
e1dcd06fd
```

Figure 11. Output of info command

Following the operations in the resulting output, it is possible to observe how the `OP_RETURN` value of the transaction, whose ID is indicated in the figure above, is computed.
The commitment operations (`append`, `prepend`, `sha256`) are consecutively executed starting from the document SHA256 hash. Actually, the subsequent SHA256 commitment operations are calculated considering the encoded value as hexadecimal, so the hash is calculated on its binary representation.

To correctly compute the result, it is necessary to execute the following Python3 code:

```
encoded = 'd1bdcd7ea02d68761710c5f52fc6f0156aca1b96093ec7f6e1f61478ebe1e70f8959f05892eed6330359086254cc0c23'
decoded = bytearray.fromhex(encoded)
sha_256_hash = hashlib.sha256(decoded)
sha_256_hash.hexdigest()
```

Figure 12. Example of SHA256 command

In the *encoded* variable it is necessary to insert the string which the SHA256 operation refers to. After all the commitments, it is necessary to execute one last SHA256 operation on the obtained string to compute a string that would be compared with the one in the `OP_RETURN` field.

To see the `OP_RETURN` value in the transaction it is possible to use the `getrawtransaction` command of the Bitcoin command line interface, followed by the transaction ID of the transaction for which it's requested to obtain the content and the value 1, which sets the verbose attribute that allows to output the content of the transaction in text format:

```
$ bitcoin-cli getrawtransaction d59099321d585e82f663fec2e06a5d0381a1056e11850c76dd4fc6727d4f32a2
```

Figure 13. Example of getrawtransaction command command


```
{
  "txid": "d59099321d585e82f663fec2e06a5d0381a1056e11850c76dd4fc6727d4f32a2",
  "hash": "d59099321d585e82f663fec2e06a5d0381a1056e11850c76dd4fc6727d4f32a2",
  "version": 1,
  "size": 197,
  "vsize": 197,
  "weight": 788,
  "locktime": 103,
  "vin": [
    {
      "txid": "94d174aa6a12c8543af898fa425701581a9b5b9f0fd313e96f44b47958da74c7",
      "vout": 0,
      "scriptSig": {
        "asm": "3044022041129dc24eef9a62f0cffd8256042b976ecd2ea048edd830d51ae6f2c84457dfd022015c219565d4d9c35c2b875fb9f031e59467638f2ab06d8955ee99773f5cf1fef[ALL]",
        "hex": "473044022041129dc24eef9a62f0cffd8256042b976ecd2ea048edd830d51ae6f2c84457dfd022015c219565d4d9c35c2b875fb9f031e59467638f2ab06d8955ee99773f5cf1fef01"
      },
      "sequence": 4294967293
    }
  ],
  "vout": [
    {
      "value": 49.99999636,
      "n": 0,
      "scriptPubKey": {
        "asm": "0 04104f9d02fe172c197fe5c4de9183978d9666c3",
        "hex": "001404104f9d02fe172c197fe5c4de9183978d9666c3",
        "reqSigs": 1,
        "type": "witness_v0_keyhash",
        "addresses": [
          "bcrt1qqsgyl8gzlctjcxtluhzdayvrj7xevekrtjfvjh"
        ]
      }
    },
    {
      "value": 0.00000000,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_RETURN f4805fc90d78912b9c083a50d08030f0649d2c0dde404231f64e2572907aa384",
        "hex": "6a20f4805fc90d78912b9c083a50d08030f0649d2c0dde404231f64e2572907aa384",
        "type": "nulldata"
      }
    }
  }
```


Figure 14. Output of getrawtransaction command

Figure 15. OP_RETURN value

If the string obtained after all the commitment operations match the `OP_RETURN` value, the integrity of the file is confirmed.

References

[1] *Decentralized Privacy-Preserving Proximity Tracing*. 2020. URL: https://github.com/DP-3T/documents/blob/master/DP3T%20White%20Paper.pdf.

[2] *DP3T-Backend-SDK*. 2020. URL: https://github.com/DP-3T/dp3t-sdk-backend.

[3] *Learning Bitcoin (and Lightning) from the Command Line*. 2020. URL: https://github.com/BlockchainCommons/Learning-Bitcoin-from-the-Command-Line.

[4] *Notarization in Blockchain*. URL: https://medium.com/@kctheservant/notarization-in-blockchain-part-1-a9795f19e28d.

[5] *OpenTimestamps*. URL: https://opentimestamps.org.

[6] *Time-Stamp Protocol (TSP)*. 2001. URL: https://tools.ietf.org/html/rfc3161.