QED

Release 1.0

Feb 27, 2020

The Project

QED is an open-source software that allows you to establish **trust relationships** by leveraging verifiable cryptographic proofs.

It can be used in multiple scenarios:

- Data transfers.
- System (or application or business) logging.
- Distributed business transactions.
- Etc.

QED **guarantees** that the system itself, even when deployed into a **non-trusted server**, cannot be modified without being detected. It also provides **verifiable cryptographic proofs** in logarithmic relation (time and size) to the number of entries.

QED is **scalable**, **resilient** and **ops friendly**:

- Designed to manage **billions of events** per shard
- Over **2000 operations per second** per shard under sustained load
- Consistent replication through RAFT
- Operable and instrumented with dozens of metrics
- **Zero config files**, fully documented single binary

CHAPTER 1

Documentation

You can find the complete documentation at: Documentation

CHAPTER 2

Project code

You can find the project code at Github

CHAPTER 3

Authors

QED was made by Hyperscale BBVA-Labs Team.

CHAPTER 4

License

QED is Open Source and available under the Apache 2 license.

CHAPTER 5

Contributions

Contributions are very welcome. See docs/source/contributing/contributing.rst or skim existing tickets to see where you could help out.

CHAPTER 6

Contents

6.1 Overview

6.1.1 What's QED

QED is an open-source software that allows you to establish **trust relationships** by leveraging verifiable cryptographic proofs.

In real-life, there are countless scenarios where maintaining a chronological record of events and operations is considered as a general principle for good internal business controls. We usually refer to this record as an audit trail, and it provides proof of compliance and operational integrity.

A paradigmatic example, with centuries of history, are the ledgers used by accountants to register all financial and non-financial data of an organization. But, the potential uses cases are not limited to that kind of information, and can be extended to any sensitive activity that could happen inside an organization, or exchanged between peers. For instance:

- Data transfers.
- System (or application or business) logging.
- Distributed business transactions.
- Etc.

Audit trails transitioned from manual to electronic records, that make this historical information more accurate, easily accessible, and usable. This also made easier the task of **auditing**, which is essential for maintaining some grade of confidence with the integrity of the stored data.

But here is where a problem of **trust** appears: how can we assure that nobody, either an insider or an outsider, tampered with such data?

QED comes to solve this lack of trust by adding **transparency** to the way that different parties interact with some specific set of data. It provides transparency by **making evident** any further non-authorized change either on such data or on the data that QED stores itself, even when deployed into a non-trusted server. And the way it achieves this capability is by using such a extended technology as **verifiable cryptographic proofs**.

6.1.2 Why

In practice, there are multiple ways to achieve a similar functionality as QED implements that range from very simple technologies, as might be the case of storing signed data (by certificate signature) into a database, to far more complicated approaches like blockchain-based technologies and smart contracts. But QED has important **advantages** over such alternatives:

- Works completely **detached** from the event source (database, logging system,...), and so from the usual way to interact with such data.
- Scales to reach **billions of events**.
- Generates proofs of **membership** or non-membership in **logarithmic time**. and with **logarithmic size**.
- Generates proofs of **temporal consistency** related to QED insertion time.

6.1.3 How

QED implements a forward-secure append-only persistent authenticated data structure. Each append operation produces as a result a cryptographic structure (a signed snapshot), which acts as a receipt for the operation, and can be used later to verify the following statements:

- Whether or not a piece of data is on QED.
- Whether or not the appended data **is consistent**, in **insertion order**, to another entry.

QED can be requested to proof whether the above statements are true or false for a specific piece of data. In response to that requests, QED returns a cryptographic proof which, combined with the original piece of data, can generate again the cryptographic value of the original snapshot.

Please refer to our *trust model* section to better understand this point.

Note: QED **does not store the data itself**, only a representation of it produced by a collision-resistant hash function.

QED **does not provide means to map a piece of data to a QED event**, so the semantic of the appended data and the relation between each item appended is also a client responsibility.

Note: This software is experimental and part of the research being done at BBVA Labs. We will eventually publish our research work, analysis and the experiments for anyone to reproduce.

6.2 Quick start

This section will guide you through QED functionality.

Mainly, you can **add events** to QED, ask for the proof that an event **has been inserted**, ask for the proof that two events are **consistent** between each other, and verify (manual or automatically) each of both proofs.

For each step we will use the **QED CLI** facility. The client will talk to the QED server and the snapshot store, so it must be configured for that proposal.

Important: To use the `qed_client` command using docker (and forget about installing golang -among other stuff-), do the following:

```
$ alias qed_client='docker run -it --net=docker_default bbvalabs/qed:v1.0.0-rc2 qed␣
↪client --endpoints http://qed_server_0:8800 --snapshot-store-url http://
↪snapshotstore:8888 --log info'
```

Don't hesitate to check the `qed_client` help facility when necessary.

```
$ qed_client -h
$ qed_client <command> -h  # Where command=(add, get, membership, incremental)
...
```

Note: In production deployments, the following variables are required, and you need to configure it. But, for this quickstart, we will use pre-defined values, so *you don't need to configure it for now*.

```
--endpoints           string  REST QED Log service endpoint list http://ip1:port1,
↪http://ip2:port2...  (default [http://127.0.0.1:8800])
--snapshot-store-url  string  REST Snapshot store service endpoint http://ip:port ␣
↪(default "http://127.0.0.1:8888")
```

6.2.1 1. Environment set up

Pre-requisites:

- **docker** (see https://docs.docker.com/v17.12/install/)
- **docker-compose** (see https://docs.docker.com/compose/install/)

Once you have these pre-requisites installed, setting up the quickstart environment is as easy as:

```
$ git clone https://github.com/BBVA/qed.git
$ git checkout v1.0.0-rc2
$ cd qed/deploy/docker
$ docker-compose up -d
```

This simple environment comprises 3 services: **QED Log server**, **QED Publisher agent**, and **Snapshot store**. You should be able to list these 3 services by typing:

```
$ docker ps
```

Once finished the Quickstart section, don't forget to clean the environment:

```
$ docker-compose down
$ unalias qed_client
```

6.2.2 2. Adding events.

In this step the client only interact with the QED server (no snapshot store info is required). The mandatory field here is the event to insert.

So, let's insert 4 simple events:

```
$ qed_client add --event "event 0"

Received snapshot with values:

    EventDigest: 5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b
    HyperDigest: 6a050f12acfc22989a7681f901a68ace8a9a3672428f8a877f4d21568123a0cb
    HistoryDigest: b8fdd4b2146fe560f94d7a48f8bb3eaf6938f7de6ac6d05bbe033787d8b71846
    Version: 0

$ qed_client add --event "event 1"
...
$ qed_client add --event "event 2"
...
$ qed_client add --event "event 3"

Received snapshot with values:

    EventDigest: 6c5cd6775eb412207f7f71f11f09047f1475b2b7526063195b777a230fe4c2a6
    HyperDigest: 7bd6cee5eb0b92801ed4ce58c54a76907221bb4e056165679977b16487e5f015
    HistoryDigest: 4f95cd9fd828abe86b092e506bbffd4662d9431c5755d68eed1ba5e5156fdb13
    Version: 3
```

Note: This operation should return only if it has been completed successfully or not. But currently it returns extra info for debugging/testing purposes.

6.2.3 3. Proof of event insertion.

3.1 Querying proof.

To get this proof we only need the original event. Therefore... has event "event 0" been inserted?

```
$ qed_client membership --event "event 0"

Querying event [ event 0 ] with latest version

Received membership proof:

    Exists: true
    Hyper audit path: <TRUNCATED>
    History audit path: <TRUNCATED>
    CurrentVersion: 3
    QueryVersion: 3
    ActualVersion: 0
    KeyDigest:␣
↪5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b
```

Yes! It was inserted in version 0 (ActualVersion), the last event inserted has version 3 (CurrentVersion), and there is a proof for you to check it.

Note: We print proofs as <TRUNCATED> due to these crypthographical proofs are too long and difficult to read.

3.2 Getting snapshots from the snapshot store.

To vefify the proof, we need to look for the right snapshot (it contains **"HyperDigest"** and **"HistoryDigest"**, the information needed to verify proofs).

```
$ qed_client get --version 3

Retreived snapshot with values:

    EventDigest:␣
↪6c5cd6775eb412207f7f71f11f09047f1475b2b7526063195b777a230fe4c2a6
    HyperDigest:␣
↪7bd6cee5eb0b92801ed4ce58c54a76907221bb4e056165679977b16487e5f015
    HistoryDigest:␣
↪4f95cd9fd828abe86b092e506bbffd4662d9431c5755d68eed1ba5e5156fdb13
    Version: 3
```

Note: The snapshot store is the right place to look for digests, instead of using the output of the step 2.

3.3 Verifying proof (manually).

Having the proof and the necessary information, let's verify the former. The interactive process will ask you to enter the info previously retrieved.

```
$ qed_client membership --event "event 0" --verify

Querying event [ event 0 ] with latest version

Received membership proof:

    Exists: true
    Hyper audit path: <TRUNCATED>
    History audit path: <TRUNCATED>
    CurrentVersion: 3
    QueryVersion: 3
    ActualVersion: 0
    KeyDigest:␣
↪5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b

Please, provide the hyperDigest for current version [ 3 ]:␣
↪7bd6cee5eb0b92801ed4ce58c54a76907221bb4e056165679977b16487e5f015
Please, provide the historyDigest for version [ 3 ] :␣
↪4f95cd9fd828abe86b092e506bbffd4662d9431c5755d68eed1ba5e5156fdb13

Verifying event with:

    EventDigest:␣
↪5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b
    HyperDigest:␣
↪7bd6cee5eb0b92801ed4ce58c54a76907221bb4e056165679977b16487e5f015
    HistoryDigest:␣
↪4f95cd9fd828abe86b092e506bbffd4662d9431c5755d68eed1ba5e5156fdb13
    Version: 3

Verify: OK
```

And yes! We can verify the membership of "event 0".

3.4 Auto-verifying proofs.

This process is similar to the previous one, but we get the snapshots from the snapshot store in a transparent way.

```
$ qed_client membership --event "event 0" --auto-verify

Querying key [ 0 ] with latest version

Received membership proof:

    Exists: true
    Hyper audit path: <TRUNCATED>
    History audit path: <TRUNCATED>
    CurrentVersion: 3
    QueryVersion: 3
    ActualVersion: 0
    KeyDigest:␣
↪5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b


Auto-Verifying event with:

    EventDigest:␣
↪5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b
    Version: 3

Verify: OK
```

6.2.4 4. Incremental proof between 2 events.

4.1 Querying proof.

For this proof we don't need the events, but the QED version in which they were added (you can get both versions by doing membership proofs as above).

```
$ qed_client incremental --start 0 --end 3

Querying incremental between versions [ 0 ] and [ 3 ]

Received incremental proof:

    Start version: 0
    End version: 3
    Incremental audit path: <TRUNCATED>
```

4.2 Getting snapshots from the snapshot store.

This process is similar to the one explained in section 2.2. As we need 2 snapshots, we repeat the query for each version.

```
$ qed_client get --version 0

Retreived snapshot with values:

    EventDigest:␣
↪5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b
    HyperDigest:␣
↪6a050f12acfc22989a7681f901a68ace8a9a3672428f8a877f4d21568123a0cb
    HistoryDigest:␣
↪b8fdd4b2146fe560f94d7a48f8bb3eaf6938f7de6ac6d05bbe033787d8b71846
    Version: 0

$ qed_client get --version 3

Retreived snapshot with values:

    EventDigest:␣
↪6c5cd6775eb412207f7f71f11f09047f1475b2b7526063195b777a230fe4c2a6
    HyperDigest:␣
↪7bd6cee5eb0b92801ed4ce58c54a76907221bb4e056165679977b16487e5f015
    HistoryDigest:␣
↪4f95cd9fd828abe86b092e506bbffd4662d9431c5755d68eed1ba5e5156fdb13
    Version: 3
```

4.3 Verifying proofs (manually).

To verify the proof manually, the process will ask you to enter the required digests.

```
$ qed_client incremental --start 0 --end 3 --verify

Querying incremental between versions [ 0 ] and [ 3 ]

Received incremental proof:

    Start version: 0
    End version: 3
    Incremental audit path: <TRUNCATED>

Please, provide the starting historyDigest for version [ 0 ]:␣
↪b8fdd4b2146fe560f94d7a48f8bb3eaf6938f7de6ac6d05bbe033787d8b71846
Please, provide the ending historyDigest for version [ 3 ] :␣
↪4f95cd9fd828abe86b092e506bbffd4662d9431c5755d68eed1ba5e5156fdb13

Verifying with snapshots:
    HistoryDigest for start version [ 0 ]:␣
↪b8fdd4b2146fe560f94d7a48f8bb3eaf6938f7de6ac6d05bbe033787d8b71846
    HistoryDigest for end version [ 3 ]:␣
↪4f95cd9fd828abe86b092e506bbffd4662d9431c5755d68eed1ba5e5156fdb13

Verify: OK
```

4.4 Auto-verifying proofs.

This process is similar to the previous one, but we get the snapshots from the snapshot store in a transparent way.

```
$ qed_client incremental --start 0 --end 3 --auto-verify

Querying incremental between versions [ 0 ] and [ 3 ]

Received incremental proof:

    Start version: 0
    End version: 3
    Incremental audit path: <TRUNCATED>


Auto-Verifying event with:

    Start: 0
    End: 3

Verify: OK
```

6.3 Trust Model

6.3.1 Description

Before starting to use QED, users need to translate their problem of trust to a more suitable conceptual model, to allow them to accurately identify which are the actors that take part in the relationship, and what are the pieces of data that must be verified.

QED defines a very simple but flexible trust model. It is composed of three main components:

- The **information** itself to which the users want to add transparency.
- A set of **actors** that interacts with the information in different ways.
- A **mapping function** that translates the information space to a univocal event that serves as input for QED.

It is clear that the information depends on the nature of the problem we are dealing with, and likewise, the mapping function definition is closely linked to it. Actors can be grouped in three categories or roles:

- Sources of information.
- Information providers.
- Information consumers.

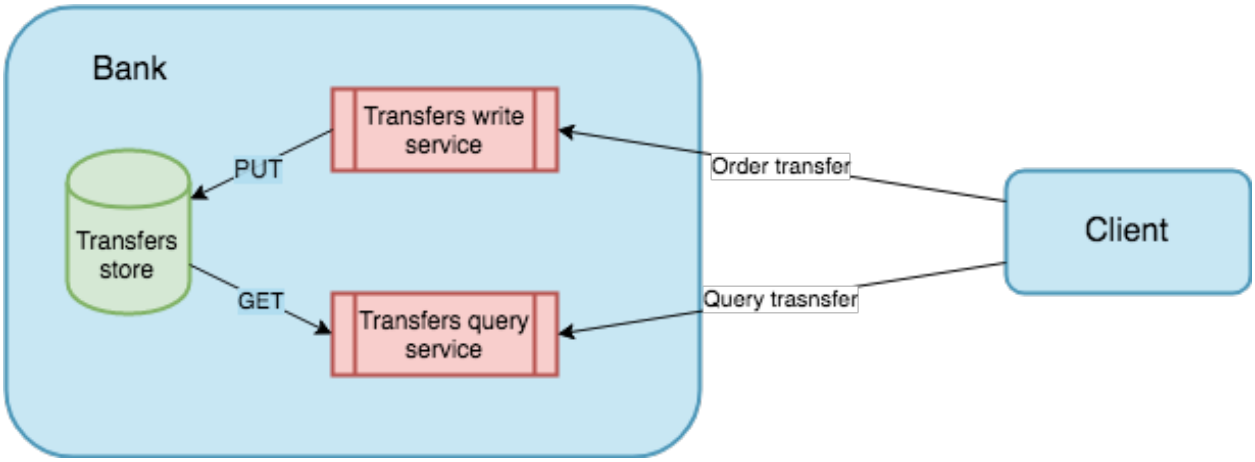
Let's see a brief example to understand better these concepts and their interactions with QED.

6.3.2 Simple scenario

Suppose a scenario where bank customers want to ensure that every money transfer related to their accounts can be verified later.

Here, the information takes the form of bank transfers which includes references to the destination accounts, a timestamp, the amount of money transferred, a concept and probably a set of different internal metadata.

The involved actors are the bank and the customer. The customer plays the role of the information consumer, and the bank plays both the roles of source and provider. Note that the bank might be divided in to different services: one for making transfers and other one for querying them.

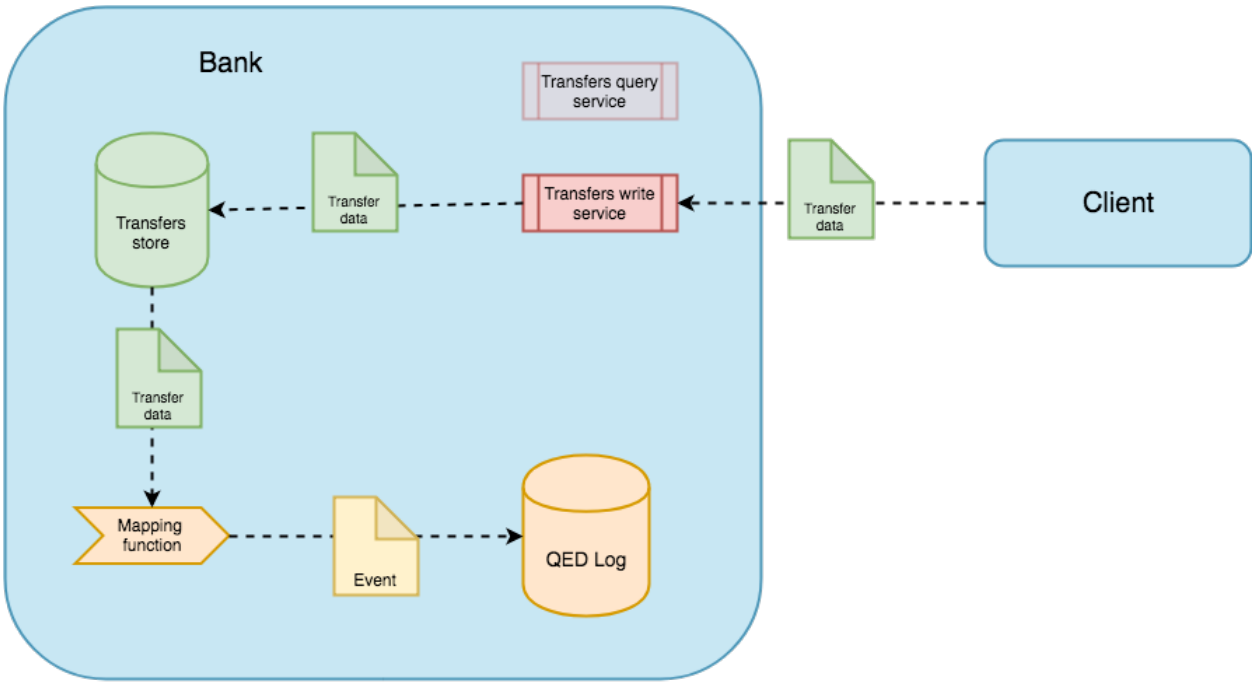


In this scenario, QED could help the provider to add transparency to its internal operations. When the client uses the bank application to order a money transfer, the application (provider) has to use the mapping function to transform every transfer data into a QED event that uniquely represents the event source entry that will be appended to the **QED Log**, which is the part of QED that stores the information needed to build the proofs. If this function has some collision, QED might not be able to issue a valid proof.

For instance, a possible event could be:

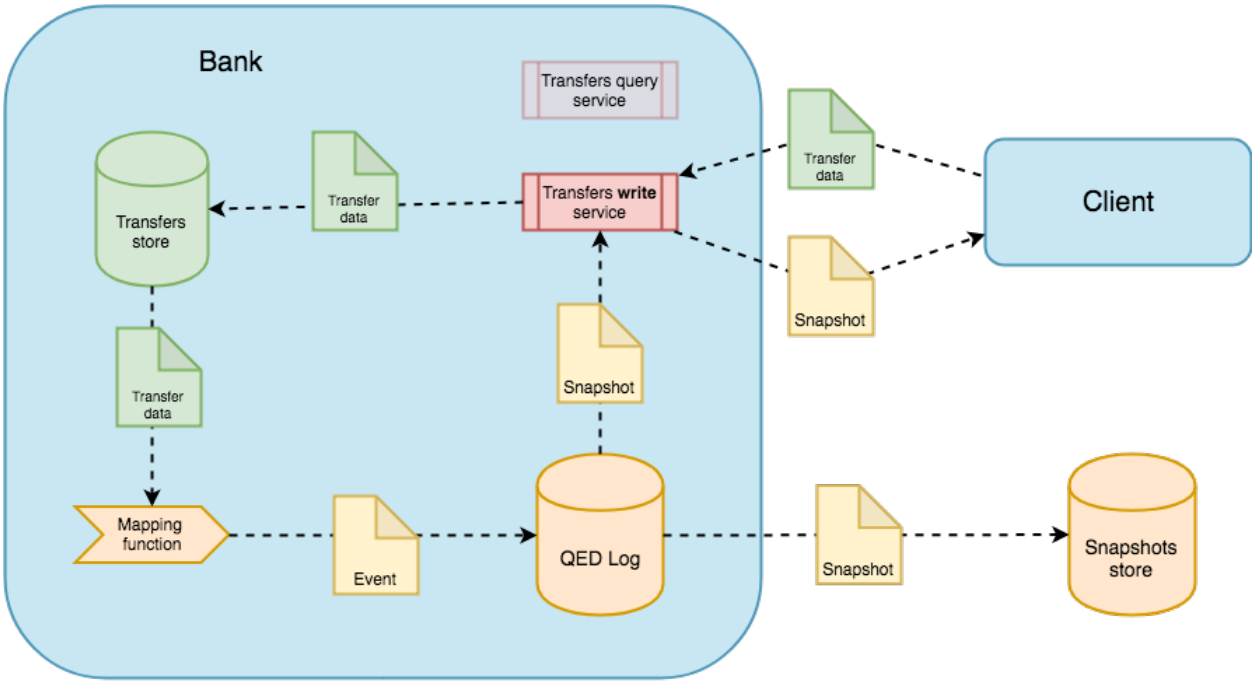
```
{
    "operation code": "money transfer",
    "user code": "0001",
    "destination": "IBAN001",
    "timestamp": 2019-05-29T10:00:35+00:00,
    "concept": "transfer money to other account",
    "amount": 1000,
    "currency": "EUR",
}
```

We necessarily have to trust this append process. If the user introduced incorrect data, he will only be able to verify such incorrect data.

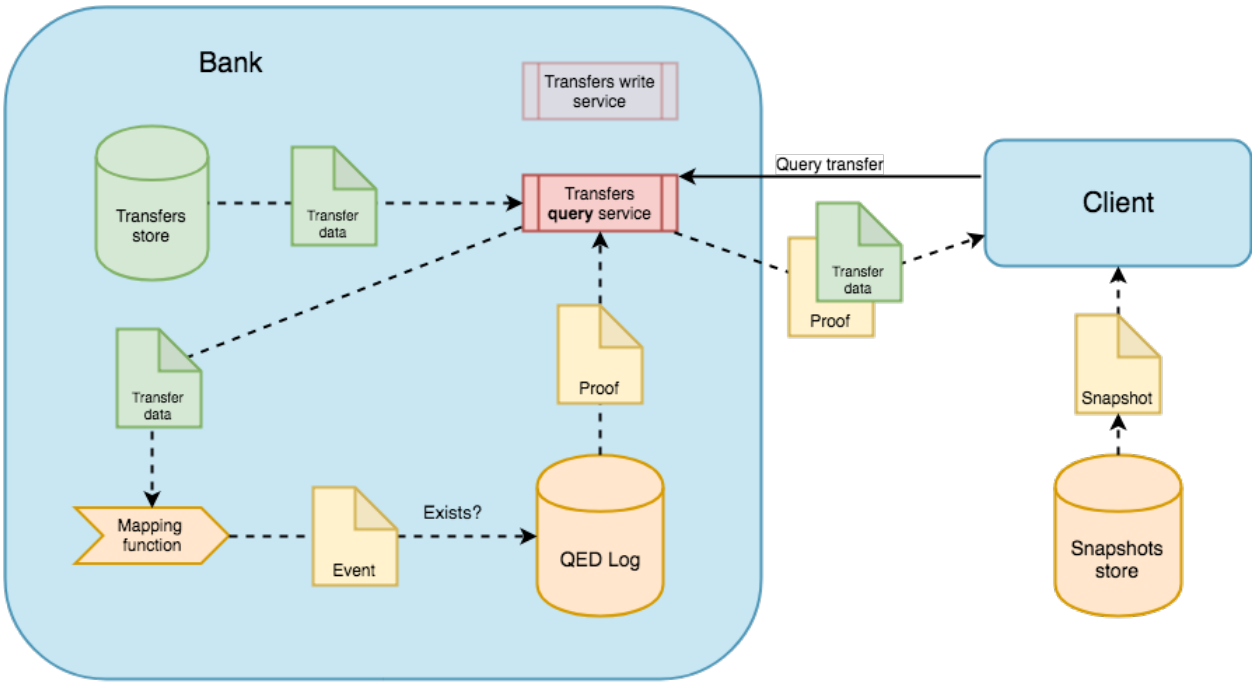


On each append operation, the QED Log will emit a signed token or receipt, called **snapshot**, that captures the full state of the log at a particular version. This snapshot will be eventually published in a (maybe public) **snaphots store** outside QED.

To provided transparency, the signed snapshot can also be delivered to the client, so he could later use it to verify the QED proof about such operation. In the same way, a third-party could be able to use that published snapshot to verify the same data. In order to avoid collusion, a snapshot store might be out of reach of the provider.



At a later time, the client might change his idea about what he did and demand the bank to proof that he ordered the transfer and/or that nobody modified the order data on its internal systems.



With a QED-based system, the bank can provide a cryptographically secure proof that verifies that:

- The QED event was effectively appended and has not been modified since then.
- The event was appended in a precise **temporal order** related to the other events from the same client. Those events can also refer to internal operations not directly related to the transfer data, allowing to verify the complete end-to-end chain of operations derived from the order.

However, the client might argue that these proofs are invalid because the QED system is deployed on bank premises (under bank control), so someone could have modified something. But, QED is also resistant to this kind of attacks, and cannot be tampered without being detected. Also, to avoid tampering, it is essential to deploy the event source, the QED Log, and the snapshot store in a way resistant to collusion.

Note: To have a deep comprehension of how QED achieves this ability, please refer to the QED's *security model* documentation.

This naive example shows the basic usage of a QED, and help to identify the fundamental components that QED requires:

- **QED Log**: where the authentication data lives.
- **Snapshot store**: that stores all snapshots generated by QED Log.
- **Events source**: the service that stores the data needed to build QED events.
- **Application**: a system/action that works with events source data.
- **Third-parties**: external actors that can use QED or be verified by QED.

There are other components which allow a QED system to be resistant to tampering:

- **QED gossip network**: a network on which QED emits snapshots.
- **QED agents**: processes subscribed to the gossip network, that execute task with the snapshot information: monitoring, auditing, etc.
- **Notification service**: notifies stakeholders of any activity of interest like alerts emitted by agents.

Note: To have a deep comprehension of these components and how they interact among them, please refer to the *architecture* documentation.

The Use cases section provides a detailed set of examples that apply the trust model to more complex scenarios.

6.4 Frequently Asked Questions

6.4.1 1. Why would anyone want to verify other's activities?

To ensure that significant information has not been modified without being noticed. For example:

- As a user of a specific service, I want to ensure the provider of such service does not change the data I produce or use. Imagine a social network that rewrites some messages they don't like. Or a compromised software download page that make users download malware to their computers.
- As a service provider, I want to ensure my user's orders and service agreements cannot be repudiated or modified after signed.
- As a service provider participating in a network of services, which operates on behalf of their customers, I don't want someone to issue orders on behalf of my customers without being noticed.

There could be hundreds of situations on which you can leverage QED's functionality to achieve tamper-evident security.

6.4.2 2. What is considered a user in QED?

A QED user is anyone who can access to the QED Log, the Snapshot Store and the QED events.

The user can be affiliated with the same organization where the QED system is deployed or with another one, or even unaffiliated, depending on the trust model you build with QED.

6.4.3 3. Can QED ensure an event is legit?

No. QED is unable to guarantee the veracity of an inserted event, it can only verify if an inserted event has not been modified and if its insertion order has not been altered.

This means that if you inserted fake events into QED, you can only be able to verify fake events.

6.4.4 4. But how can QED help me to achieve that guarantees?

QED provides cryptographic proofs that demonstrate:

- Whether or not a piece of data was inserted in QED.
- Whether or not the appended data is consistent, in its insertion order to another entry.

4.1 And with only those two proofs, can we achieve all those functionalities?

No, QED is a software that helps you to implement processes and other pieces of software which will enable you to build those capabilities.

6.4.5 5. How is QED different from digital signatures or blockchain?

In the following table we compare the main characteristics of each technology:

Feature	Digital signature + DB	Blockchain	QED
Prove inclusion (time)	Logarithmic	Linear	Logarithmic
Prove non-inclusion (time)	Linear	Linear	Logarithmic
Prove append-only (time)	Linear	Linear	Logarithmic (non deletion proof can take linear time)
Consistency proof size	Linear	Linear	Logarithmic
Proof size	Constant	Constant	Logarithmic
Tampering detection (time)	No, if the PK gets compromised	Linear	Logarithmic

Either entries digitally signed in a database or a blockchain network can provide for proofs like QED, and similar functionality can be achieved by all of them.

QED really shines when it is used to build a lot of inclusion proofs or consistency proofs, because its performance allows you to save a lot of space and computing power, which can be transformed into scalability.

QED is designed to handle **billions of entries** at over 2000 operations per second.

6.4.6 6. Is it secure?

The security model of QED is based on three pillars:

- Strong cryptographic hash functions (SHA256 or BLAKE2).
- Separated source data stores from the proof store and the snapshot store.
- Active and decentralized monitoring.

It is fundamental for QED to use a fast, reliable and unbroken hash function. This allows you to avoid collisions and ensure event information cannot leak.

Also, in order to verify any of the QED issued proofs a user will need three items:

- The original event inserted into QED.
- The proof issued by QED.
- The authentication token (snapshot) published by QED when the event was inserted.

And lastly, QED includes active monitoring that throw alerts if something goes wrong.

But be aware, we intrinsically trust the append operation to QED. If you insert fake data, you verify fake data. There is no way to fully prevent that, in this system or in any other.

6.4.7 7. Will QED alert me from changes or tampering attempts?

No. QED will never issue proofs proactively nor be aware of tampering. It is the user responsibility to actively monitor QED to detect those modification attempts.

Besides, QED will also help you detect tampering in itself.

6.4.8 8. Is QED a data store? Can I save my data into QED to secure it?

No. QED does not store any data, it only stores a fingerprint of such data using a strong hashing function. It only supports three operations:

- Append a new entry.
- Ask for an inclusion proof.
- Ask for a consistency proof.

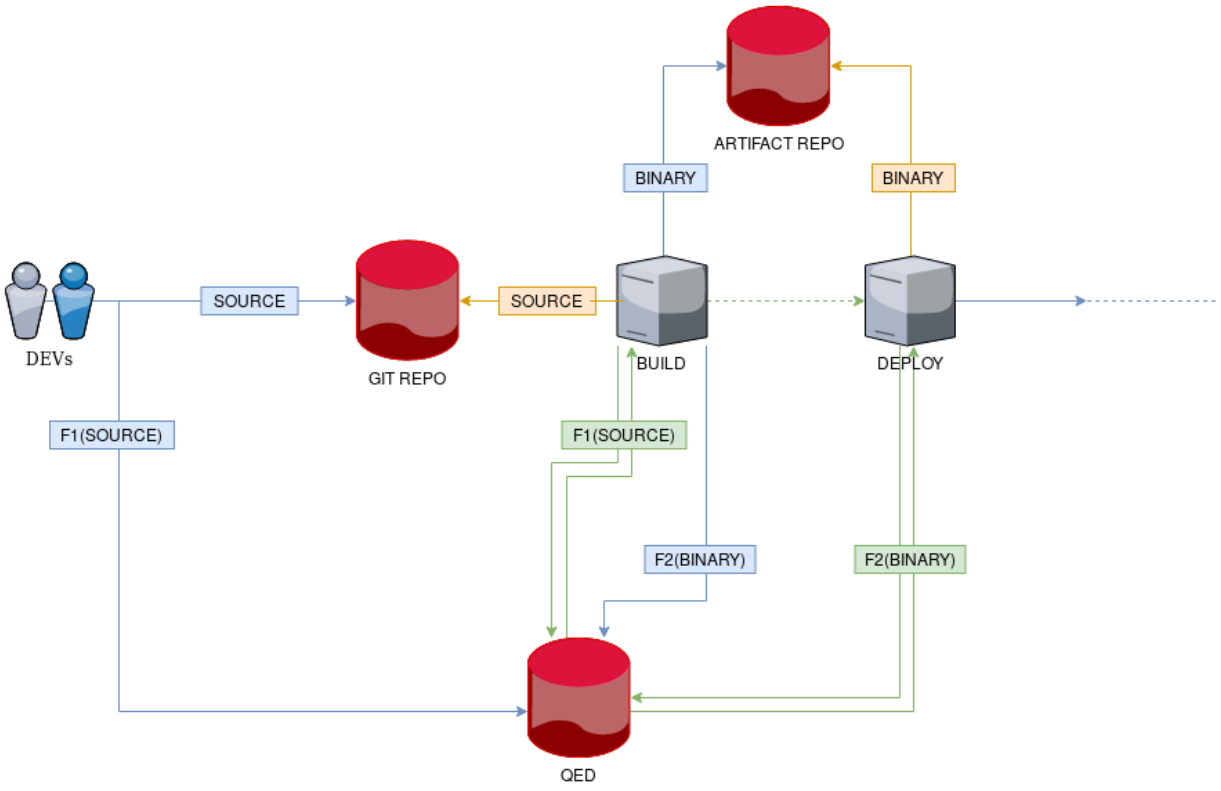
6.5 Commit certification

In this use case we will show how to add transparency to a simple software deployment pipeline, starting from the source code a developer commits to a source repository, and ending with the deployment of the corresponding built artifact.

This way, the developer can ensure that what he intented to publish is what it was finally deployed.

6.5.1 Theory and operation

In order to add transparency to the process we will need to identify firstly what are the elements of our trust problem and then try to adapt them to the components defined in our *QED's trust model*: information, actors and mapping function(s).



As we can see from the figure, there are two kinds of information to which we need to add transparency: the original source code commited by the developer and the binary artifact built by the CI tool.

In this case, the actors are multiple (developer, repositories and pipeline processes) and some of them take different roles depending on the step of the pipeline being executed.

Let's explain the process in detail.

First step: source committing

We have an actor, the **developer**, that takes the role of source of information. He makes some changes to the source code and commits them to the Git repository. The repository will therefore be the infomation provider in our trust model and the first component we want to add transparency to. Every consumer of that repository will need some kind of proof to verify its integrity.

To achive this, the developer can use a particular mapping function `F1` that translates the resulting source code to a unique QED event. But first, we need to identify what makes it unique.

For this event, the original commit hash and a SHA256 digest of all files (excluding the .git folder) will provide a concise information that will vary whenever even a single character gets changed.

Note: `F1` output example:

```
{
    "commit_hash": "4b1a0b7be7b5982dc778e76adacbb6348632ff4d",
    "src_hash": "b9261acdcc979434d37ed8211ad6014309752cb6a02705a40dc8dbaf9cdcd89b",
}
```

Then, the developer can take the event resulting after applying the function to the source code `F1(SOURCE)`, and insert it into the QED Log.

Second step: artifact building

Once the source code has been committed to the repository, a hook fires the **build** phase of the pipeline which downloads the source code and generates a binary artifact. The build process acts here as the consumer actor in the trust model and thus, needs to have confidence in the integrity of the repository.

To do that, it could use the same mapping function `F1` to generate again the QED event and then request a membership query to the QED Log. With the resulting cryptographic proofs and the QED event, it could verify the original information, the source code, as valid.

Third step: uploading artifact

Now, the build process comes from acting as a consumer to take the role of source of information. The binary artifact is now the information we want to verify and the artifact repository becomes the new information provider.

Thus, the build process has to use a new mapping function `F2` to translate the resulting artifact to a unique QED event `F2(BINARY)`, and then, insert such event into the QED Log.

For this function, the SHA256 digest of the binary file, will be simple and good to detect changes.

Note: `F2` output example:

```
{
    "artifact_hash": "pcdcc979434d37e4b1a0b4309752cb6a0277c778e76adacbb6348632ff4d",
}
```

Fourth step: artifact deployment

Once the binary artifact has been uploaded to the artifact repository, a new hook fires the **deploy** phase of the pipeline which downloads the binary file and deploys it to the corresponding environment. Now, the deploy process acts as the consumer actor in the trust model that needs to have confidence in the integrity of the artifact repository.

To achieve that, it must use the same mapping function `F2` to generate the corresponding QED event in order to request a membership proof from the QED Log. Again, combining the resulting cryptographic proofs with the QED event, the process could verify the original information as valid.

6.5.2 Working example

Adding transparency to a GIT repository

Warning: The following snippets assume a working QED installation. Please refer to the *Quick start* page.

The following snippet simulates the creation of a QED event starting from the source code recently committed. As mentioned before, we are using the **commit_hash** and the **source_hash** as the output of the mapping function `F1(SOURCE)` to unambiguously identify a source code.

```
# Create the source code event
commit_hash=$(git rev-parse HEAD)
src_hash=$(echo $(find . -type f -not -path "./.git/*" -exec sha256sum {} \; | sort -
↪k2) | sha256sum | cut -d' ' -f1)
cat > event.json <<EOF
{
    "commit_hash": "${commit_hash}",
    "src_hash": "${src_hash}",
}
EOF
```

Alongside pushing the code to the git repo, the developer (or a githook) adds the event to the QED Log.

```
# pushing the event to QED server
qed_client \
    add \
    --event "$(cat event.json)"
```

Once the QED stores the event, the `BUILD` stage will fetch the source code from the git repo and, just before building the binary artifact, generate again the QED event to request a membership proof to QED Log. After verifying the integrity of the source code at the repository, it will continue with the next step.

```
# Verify the proof
# please note the --auto-verify flag, without this flag the operation
# will returns the cryptographic proof
qed_client \
    membership \
    --event "$(cat event.json)" \
    --auto-verify
```

Adding transparency to the artifacts repository

Once the BUILD stage creates the `BINARY` file, it applies the mapping function `F2(BINARY)` to the file and obtains a new QED event.

```
# Create the artifact event
artifact_hash=$(sha256sum archived/gin | cut -d' ' -f1 )
cat > bin_event.json <<EOF
{
    "artifact_hash": "${artifact_hash}",
}
EOF
```

Alongside pushing the binary artifact to the repository it adds the event to the QED Log. As you can see, there is a repeating pattern of `source -> [QED|Untrusted-source] <- auditor` in the way QED creates the transparency.

```
# pushing the artifact event to QED server
qed_client \
    add \
    --event "$(cat bin_event.json)"
```

And finally, the DEPLOY stage can request again a proof from the QED Log and verify the integrity of the artifact before deploying it.

```
# Verify the proof
qed_client \
    membership \
    --event "$(cat bin_event.json)" \
    --auto-verify
```

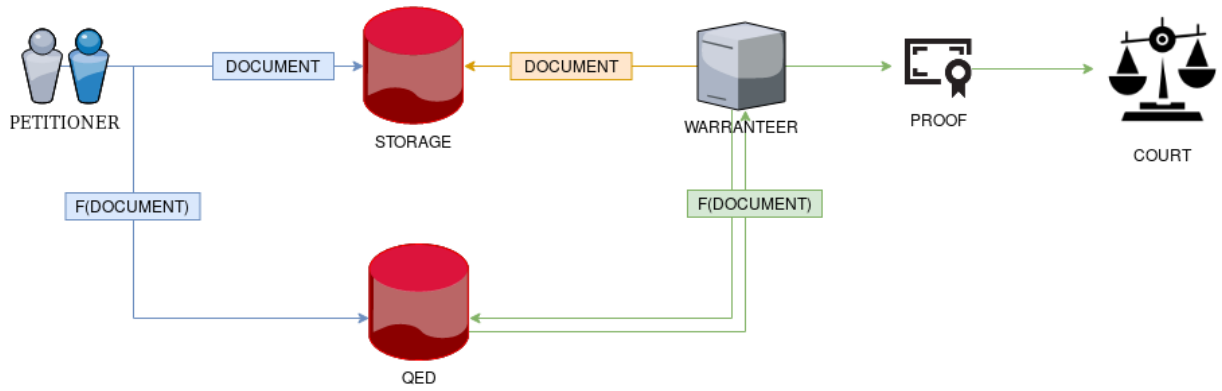
6.6 Certification of Documents, Emails, Agreements, etc.

In this use case we will show how to add transparency to a particular transaction or agreement that got captured in a `DOCUMENT`, by allowing the issuer to certify that the document has not been altered.

6.6.1 Theory and Operation

Tip: For the sake of clarity, **Document** is anything that could be suitable to keep track of the original transaction, such as Emails, Agreements, Dues, etc…

First of all, we need to identify what are the elements of the problem to address and how we can adapt them to the components defined in our *QED's trust model*: information, actors and mapping function(s).



As we can see from the figure, the information we want to add transparency to, is the `DOCUMENT` itself, which gets inserted in a particular `STORAGE`. This storage acts as the **information provider**, and it can be considered as untrusted.

The `PETITIONER` is the actor interested in keeping track of the contents of the document, so he takes the role of **source of information** and inserts the document into the storage.

Simultaneously, he uses a mapping function `F` to translate the information to a unique QED event `F(DOCUMENT)`. He could use the SHA256 digest of the contents of the document.

Note: `F` output example:

```
{
    "digest": "4b1a0b7be7b5982dc778e76adacbb6348632ff4d",
}
```

Now, suppose there is a court trial that demands proofs of integrity to the entity in charge of keeping the document, the one we have called `WARRANTEER`. This actor also have to act as the **information consumer** in the trust model, and thus, needs to have confidence in the integrity of the storage.

To do that, it could use the same mapping function `F` to generate again the QED event and then, ask for a membership proof to the QED Log. Combining the resulting cryptographic proofs with the QED event, the `WARRANTEER` could verify the original information as valid.

6.6.2 Working example

> **Warning:** The following snippets assume a working QED installation. Please refer to the *Quick start* page.

The following snippet simulates the creation of a QED event starting from the `DOCUMENT` recently emitted. As mentioned before, we are using the SHA256 digest of the contents of the file as the output of the mapping function `F1(DOCUMENT)` to unambiguously identify the document.

```
# Create the document event
document_hash=$(sha256sum <document> | cut -d' ' -f1 )
cat > document_event.json <<EOF
{
    "document_hash": "${document_hash}",
}
EOF
```

Alongside inserting the document into the storage, we add the event to the QED Log.

```
# pushing the document event to QED server
qed_client \
    add \
    --event "$(cat document_event.json)"
```

Finally, we can generate again the QED event to request a membership proof from QED Log and verify the proof.

```
# Verify the proof
qed_client \
    membership \
    --event "$(cat document_event.json)" \
    --auto-verify
```

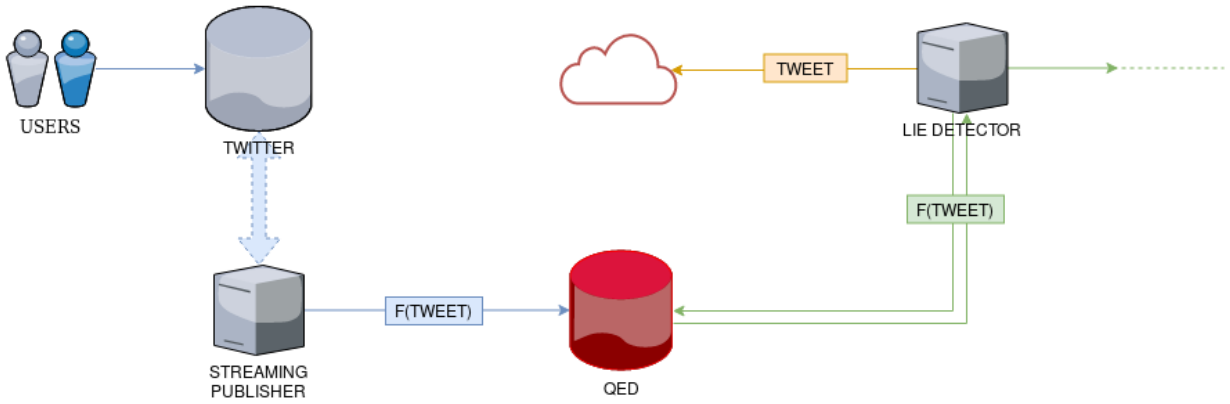
6.7 Lie Detector for Tweeter feeds

Nowadays, with the boom of fake news, it could be interesting to detect inconsistencies between the contents that were originally published in a particular media and what is currently accessible to the public.

In this use case, we will show how to add transparency to messages published in a social network like Twitter, by allowing the users to verify that already published tweets have not been altered.

6.7.1 Theory and Operation

First of all, we need to identify what are the elements of the problem to address and how we can adapt them to the components defined in our *QED's trust model*: information, actors and mapping function(s).



As we can see from the figure, the information we want to add transparency to, is the set of tweets published by one or multiple users. Those users are the actors interested in keeping track of the contents of their own publications, so they take the role of **sources of information**.

The tweets get inserted into the internal storage operated by Twitter, Inc. This storage acts as the **information provider** in out trust model and, by definition, is considered unstrusted. The way users interact with this provider is through its public API.

In order to push events to QED, a tool like a `STREAMING PUBLISHER` becames necessary to drain messages from Twitter.

Note: See golang go-twitter module, python's tweeepy library, or npm twitter package *streaming-api* capabilities, to create your own tool.

Such `STREAMING PUBLISHER` tool would use a mapping function `F` to translate the tweets contents to a unique QED event `F(TWEET)`. Tweets data and metadata (like username, date and text) could serve to identify each tweet unambiguously.

Note: `F` output example:

```
{
    "user_screen_name": "TwitterDev",
    "date": "22:01 - 6 may. 2019",
    "text": "Today's new update means that you can finally add Pizza Cat to␣
↪your Retweet with comments! Learn more about this ne... https://t.co/
↪Rbc9TF2s5X",
}
```

Finally, the `LIE DETECTOR` service would act as the **information consumer** in the trust model, and will audit the information provided by Twitter's public APIs.

To do that, it could use the same mapping function `F` to generate again the QED event and then, ask for a membership proof to the QED Log. Combining the resulting cryptographic proofs with the QED event, the `LIE DETECTOR` could verify the original information as valid.

6.7.2 Working example

Warning: The following snippets assume a working QED installation. Please refer to the *Quick start* page.

The following snippet simulates the creation of a QED event starting from a particular tweet recently published. As mentioned before, we are applying a mapping function `F(TWEET)` to some data and metadata from the tweet.

```
# Create the tweet event
$ cat > tweet_event.json <<EOF
{
    "user_screen_name": "TwitterDev",
    "date": "22:01 - 6 may. 2019",
    "text": "Today's new update means that you can finally add Pizza Cat to your␣
↪Retweet with comments! Learn more about this ne... https://t.co/Rbc9TF2s5X",
}
EOF
```

Then, we insert the event into QED Log:

```
# pushing the tweet event to QED server
qed_client \
    add    \
    --event "$(cat tweet_event.json)"
```

Finally, we can generate again the QED event to request a membership proof from QED Log and verify the proof.

```
# Verify the proof
qed_client                          \
    membership                      \
    --event "$(cat tweet_event.json)" \
    --auto-verify
```

6.8 Architecture and components

TODO

6.9 How does it works (long version)

TODO

6.10 Security Model

TODO

6.11 Glossary

The purpose of this section is to equip the reader with necessary background about the most common keywords and concepts used in the development of verifiable (or authenticated) data structures.

6.11.1 Cryptographic primitives

Cryptographic hash functions and digital signatures are the fundamental building blocks for creating authenticated data structures.

Hash functions

A cryptographic hash function compresses an arbitrary large message *m* into a fixed size digest *h*. Due to the large space of messages mapped, collisions are inevitable but they must be computationally hard to find. A cryptographic hash function must conform with the following properties:

- **Preimage resistance:** given a digest **h* <- H(*m*)* for message *m*, it must be computationally hard to find a preimage *m'* generating *h* without knowledge of *m*.
- **Second preimage resistance:** given a fixed preimage *m*, it must be computationally hard to find another preimage *m' != m* such that *H(*m*) = H(*m'*)*.
- **Collision resistance:** it must be computationally hard to find any distinct preimages *m1* and *m2* such that *H(*m1*) = H(*m2*)*.

6.11.2 Digital signatures

A digital signature is a mathematical scheme for demonstrating the *authenticity*, *non-repudiation* and *integrity* of a message. So a valid digital signature gives a recipient a reason to believe that the message was created by a known sender, that the sender cannot deny having sent the message and that the message was not altered in transit.

6.11.3 Tree-based data structures

A tree is an (un)ordered collection of entities, not necessarily unique, that has a hierarchical parent-child relationship between pairs of entities. Every tree has a single *root* node designating the start of the tree, and each *descendant* is recursively defined as a tree. A node is said to be an *ancestor* to all its descendants, and a parent to its concrete *children*. All children that have the same parent are referred to as *siblings*, and every node without children is referred to as a *leaf*. The root is said to be found at *level* one, the *height *is the number of levels in the tree, and the *depth* of a subtree rooted at a leaf is zero.

Binary tree

A binary tree is a tree where each node is restricted to at most a *left child* and a *right child*.

Perfect binary tree

A binary tree of height *h* that must contain exactly 2^h - 1 nodes.

Full binary tree

A binary tree which all nodes must have two or no children.

Complete binary tree

A binary tree which must be filled left-to-right at the lowest level, and entirely at the level above.

6.11.4 Merkle tree

A binary tree that stores values at the lowest level of the tree and uses cryptographic hash functions. While leaves compute the hash of their own attributes, parents derive the hash of their children's hashes concatenated left-to-right. Therefore the hash rooted at a particular subtree is recursively dependent on all its descendants, effectively serving as a succinct summary for that subtree.

Membership proof

A Merkle tree can prove values to be present by constructing efficient *membership proofs*. Each proof must include a *Merkle audit path*, and it is verified by recomputing all hashes, bottom up, from the leaf that the proof concerns towards the root. The proof is believed to be valid if the recomputed root hash matches that of the original Merkle tree, but to be convincing it requires a trustworthy root (e.g., signed by a trusted party or published periodically in a newspaper).

Merkle audit path

A Merkle audit path for a leaf is the list of all additional nodes in the Merkle tree required to compute the Merkle Tree Hash for that tree. If the root computed from the audit path matches the true root, then the audit path is proof that the leaf exists in the tree.

6.11.5 History tree

An append-only Merkle tree that stores *events* left-to-right at the lowest level of the tree. It is not lexicographically sorted, and unable to generate efficient non-membership proofs, but it is *naturally persistent*, supports efficient membership proofs and allows to generate *incremental proofs*.

Persistent nature

A history tree is naturally persistent, in the sense that past versions of the tree can be efficiently reconstructed and queried for membership.

Incremental proof

A history tree can show consistency between root hashes for different views, and that requires proving all events in the earlier view present in the newer view. It is achieved by returning just enough information to reconstruct both root hashes checking if expected roots are obtained.

6.11.6 Binary search tree

A binary tree that requires the value of each node to be greater (or lesser) that the value of its left (or right) child. This property, referred to as the *BST property*, implies a lexicographical order and allows every look-up operation to use a divide-and-conquer technique known as *binary search*. Because the time required to complete a binary search is bounded by the height of the BST, it is important that the tree structure remains *balanced*.

6.11.7 Heap

A specialized tree-based data structure used in the context of priority queues. It associates each node a priority and preserves, at all times, two properties: the *shape property*, requiring that the heap is a complete binary tree; and the *heap property*, requiring that every node has a lower or equal priority with respect to its parent.

6.11.8 Treap

A randomized search tree associating with each entity a *key* and a randomly selected priority. Treaps enforce the BST property with respect to keys, the heap property with respect to priorities, and are also *set-unique*. Set-uniqueness ensures the tree structures of identical collections to be equivalent, thereby implying *history independence* if priorities are assigned deterministically.

6.11.9 Hash treap

A lexicographically sorted history independent key-value store combining a regular Merkle tree and a deterministic treap. Each node is associated with an entity and every (non-)member has a unique position, therefore hash treaps support efficient (non-)membership proofs.

6.11.10 Sparse Merkle tree

A Merkle tree which depth is fixed in advance with respect to the underlying hash function H, meaning there are always 2^|H(.)| leaves. These are referred left-to-right by indices, and are associated with either *default* or *non-default* values. In the latter case the hash of a key determines the index, which implies there is a unique leaf reserved for every conceivable digest H(*k*). This allows generation of (non-)membership proofs using regular Merkle audit paths. The SMT is *sparse* because the large majority of all leaves will be empty, and consequently most nodes rooted at lower levels of the tree derive identical default hashes.

6.12 Cluster mode

This section will guide you through QED cluster features.

Here, you will **check cluster information**, **add events**, and **query proofs** in a cluster environment (against more than one QED server).

For this functionality we will use the **QED CLI** facility. The client will talk to the QED servers, so it must be configured for that proposal.

Important: To use `qed_client` command using docker (and forget about installing golang -among other stuff-), do the following:

```
$ alias qed_client='docker run -it --net=docker_default bbvalabs/qed:v1.0.0-rc2 qed␣
↪client --log info'
```

Don't hesitate to check `qed_client` help command when necessary.

6.12.1 1. Environment set up

Pre-requisites:

- **docker** (see https://docs.docker.com/v17.12/install/)
- **docker-compose** (see https://docs.docker.com/compose/install/)

Once you have these pre-requisites installed, setting up the required environment is as easy as:

```
$ git clone https://github.com/BBVA/qed.git
$ cd qed/deploy/docker
$ docker-compose -f cluster-mode.yml up -d
```

This environment comprises three **QED Log server** services: **qed_server_0** will be the cluster leader, while **qed_server_1** and **qed_server_2** will be followers. You should be able to list these service by typing:

```
$ docker ps
```

Once finished the cluster-mode section, don't forget to clean the environment:

```
$ docker-compose -f cluster-mode.yml down
$ unalias qed_client
```

6.12.2 2. Checking cluster information.

QED servers have a shard information endpoint that returns how is the cluster formed. Here we use **curl** to ask for this information, since there is no command for this.

Here we will ask **qed_server_0** (notice that "nodeID: server0"), but you can try another nodes:

```
$ curl -sS -H "Api-key:my-key" http://localhost:8800/info/shards | python -m json.tool

{
    "nodeId": "server0",
    "leaderId": "server0",
    "uriScheme": "http",
    "shards": {
        "server0": {
            "nodeId": "server0",
            "httpAddr": "qed_server_0:8800"
        },
        "server1": {
            "nodeId": "server1",
            "httpAddr": "qed_server_1:8800"
        },
        "server2": {
            "nodeId": "server2",
            "httpAddr": "qed_server_2:8800"
```

(continues on next page)

(continued from previous page)

```
        }
    }
}

$ curl -sS -H "Api-key:my-key" http://localhost:8801/info/shards | python -m json.
↪tool  # For qed_server_1
$ curl -sS -H "Api-key:my-key" http://localhost:8802/info/shards | python -m json.
↪tool  # For qed_server_2
```

Servers information is shared between QED servers via Raft. Once a server joins the cluster (vía cluster leader), it shares its information and receive others. Servers interchange information also when a server leaves the cluster, or when leader changes.

6.12.3 3. Adding events.

Only QED cluster leader accepts insertions.

qed_client is configured by default to discover the cluster topology (using the above information), identify which server is the cluster leader, and send requests directly to this server.

```
$ qed_client --endpoints http://qed_server_0:8800,http://qed_server_1:8800,http://qed_
↪server_2:8800 add --event "event 0"

Received snapshot with values:

EventDigest: 5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b
HistoryDigest: b8fdd4b2146fe560f94d7a48f8bb3eaf6938f7de6ac6d05bbe033787d8b71846
HyperDigest: 6a050f12acfc22989a7681f901a68ace8a9a3672428f8a877f4d21568123a0cb
Version: 0
```

Notice that given just 1 endpoint is enough to discover the cluster topology, and the cluster leader (qed_server_0).

```
$ qed_client --endpoints http://qed_server_1:8800 add --event "event 1"

$ qed_client --endpoints http://qed_server_2:8800 add --event "event 2"
```

6.12.4 4. Querying membership proof.

Proofs can be asked to any cluster member.

```
$ qed_client --endpoints http://qed_server_0:8800 membership --event "event 0"

Querying key [ event 0 ] with latest version

Received membership proof:

Exists: true
Hyper audit path: <TRUNCATED>
History audit path: <TRUNCATED>
CurrentVersion: 2
QueryVersion: 2
ActualVersion: 2
KeyDigest: 5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b
```

(continues on next page)

(continued from previous page)

```
$ qed_client --endpoints http://qed_server_1:8800 membership --event "event 0"
$ qed_client --endpoints http://qed_server_2:8800 membership --event "event 0"
```

6.12.5 5. Shutting down a server

Here we will stop the QED cluster leader to force a leader election.

```
$ docker stop qed_server_0
```

6.12.6 6. Repeat steps 2-4 several times.

Check that shard information has been modified, and remember that **qed_server_0** will not work.

6.13 Backup and Restore

This section will guide you through QED backup and restore functionalities.

6.13.1 Backup

Here, you will **create backups**, **list backups**, and **delete backups** by interacting to the QED management API (not the same API as the one used in the QuickStart section).

For the backup functionality we will use the backup **QED CLI** facility. The backup client will talk to the QED server, so it must be configured for that proposal. To **add events**, we will use the same client as in QuickStart.

Important: To use `qed_backup` and `qed_client` command using docker (and forget about installing golang -among other stuff-), do the following:

```
$ alias qed_client='docker run -it --net=docker_default bbvalabs/qed:v1.0.0-rc2 qed␣
↪client --endpoints http://qed_server_0:8800 --snapshot-store-url http://
↪snapshotstore:8888 --log info'

$ alias qed_backup='docker run -it --net=docker_default bbvalabs/qed:v1.0.0-rc2 qed␣
↪backup --endpoint http://qed_server_0:8700 --log info'
```

Don't hesitate to check both `qed_backup` and `qed_client` help commands when necessary.

```
$ qed_backup -h
$ qed_backup <command> -h  # Where command=(create, list, delete)
...
```

---

1. Environment set up

Pre-requisites:

- **docker** (see https://docs.docker.com/v17.12/install/)
- **docker-compose** (see https://docs.docker.com/compose/install/)

Once you have these pre-requisites installed, setting up the required environment is as easy as:

```
$ git clone https://github.com/BBVA/qed.git
$ cd qed/deploy/docker
$ docker-compose -f backup-restore.yml up -d
```

This environment is not similar to the QuickStart's one. It comprises 1 service: **QED Log server**. To test backup/restore functionality we do not need any other service but this one. Moreover, now the DB folder of Qed Log server is mapped to a host temporal folder, to be used later in the restore section. You should be able to list this service by typing:

```
$ docker ps
```

Once finished the backup&restore section, don't forget to clean the environment:

```
$ docker-compose -f backup-restore.yml down
$ unalias qed_client
$ unalias qed_backup
```

2. Adding events.

Similarly to QuickStart guide, let's insert 2 events:

```
$ for i in {0..1}; do qed_client add --event "event $i"; done

Received snapshot with values:

EventDigest: 5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b
HyperDigest: 6a050f12acfc22989a7681f901a68ace8a9a3672428f8a877f4d21568123a0cb
HistoryDigest: b8fdd4b2146fe560f94d7a48f8bb3eaf6938f7de6ac6d05bbe033787d8b71846
Version: 0


Received snapshot with values:

EventDigest: fb378474af5953bec611fcb2602c5b61271c1f233b60c0adba76d5d6f47a50c4
HyperDigest: 15814ee2f820da9c126fc740d5b4de034d250a3f5fe6e58ab5616026cb65b3dd
HistoryDigest: ae6fe0b70e09b12eeea3bc2cb923d239d184a2b30a578e201ad952e2e9a405f2
Version: 1
```

3. Creating backups.

```
$ qed_backup create

Backup created!
```

The version of the last inserted event is stored into the backup metadata.

4. Listing backups.

```
$ qed_backup list
```

(continues on next page)

(continued from previous page)

```
Backup list:
Id: 1       Timestamp: 2019-07-17T13:13:26  Version: 1      Size(GB): 0     Num.
↪Files: 4
```

5. Repeat steps 2-4 several times.

```
$ for i in {2..3}; do qed_client add --event "event $i"; done
$ qed_backup create
$ for i in {4..5}; do qed_client add --event "event $i"; done
$ qed_backup create
$ qed_backup list

Backup list:
Id: 1       Timestamp: 2019-07-17T13:13:26  Version: 1      Size(GB): 0     Num.
↪Files: 4
Id: 2       Timestamp: 2019-07-17T13:13:40  Version: 3      Size(GB): 0     Num.
↪Files: 4
Id: 3       Timestamp: 2019-07-17T13:13:54  Version: 5      Size(GB): 0     Num.
↪Files: 4
```

6. Deleting backups.

```
$ qed_backup delete --backup-id=1

Backup deleted!

$ qed_backup list

Backup list:
Id: 2       Timestamp: 2019-07-17T13:13:40  Version: 3      Size(GB): 0     Num.
↪Files: 4
Id: 3       Timestamp: 2019-07-17T13:13:54  Version: 5      Size(GB): 0     Num.
↪Files: 4
```

6.13.2 Restore

Here, you just will **restore** a QED log server state **from a previous backup**, being able to choose the latest backup (by default) or a certain backup ID to recover from (see IDs above).

1. Environment set up.

To simulate a new QED log server, let's destroy the current environment and create a new one from scratch. To destroy the environment, just do:

```
$ cd qed/deploy/docker
$ docker-compose down
...
```

Remember that we saved the backups folder in a host path. So let's check that the folder has backup information.

```
$ tree /tmp/backups/

/tmp/backups/
├── meta
│   ├── 2
│   └── 3
├── private
│   ├── 2
│   │   ├── 000003.log
│   │   ├── CURRENT
│   │   ├── MANIFEST-000004
│   │   └── OPTIONS-000014
│   └── 3
│       ├── 000003.log
│       ├── CURRENT
│       ├── MANIFEST-000004
│       └── OPTIONS-000014
└── shared
```

There are information of backups 2 and 3 as expected (we deleted backup 1 before).

To create a new environment from scratch, just do:

```
$ docker-compose -f backup-restore.yml up -d
```

Finally, let's check that the "event 0" is not present in the new QED log server.

```
$ qed_client membership --event "event 0"

Querying event [ event 0 ] with latest version

Received membership proof:

    Exists: false
    Hyper audit path: <TRUNCATED>
    History audit path: <TRUNCATED>
    CurrentVersion: 18446744073709551615
    QueryVersion: 18446744073709551615
    ActualVersion: 18446744073709551615
    KeyDigest: 5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b
```

Notice that the event does not exist.

2. Restore process.

Get into the QED log server:

```
$ docker exec -it qed_server_0 /bin/bash
```

Restore backup 2, from the interal docker backup folder, to the interal docker path where the DB is:

```
$ qed restore --backup-dir "/var/tmp/qed0/db/backups/" --restore-path "/var/tmp/qed0/
↪db/" --backup-id 2 --log info
```

Exit the QED server, and restart the container to make QED server aware of the restored DB.

```
$ exit
$ docker restart qed_server_0
```

3. Check event membersip.

Event 0 (and up to event 3) should be there:

```
$ qed_client membership --event "event 0"

Querying key [ event 0 ] with latest version

Received membership proof:

Exists: true
Hyper audit path: <TRUNCATED>
History audit path: <TRUNCATED>
CurrentVersion: 3
QueryVersion: 3
ActualVersion: 0
KeyDigest: 5beeaf427ee0bfcd1a7b6f63010f2745110cf23ae088b859275cd0aad369561b
```

But event 4 should not:

```
$ qed_client membership --event "event 4"

Querying key [ event 4 ] with latest version

Received membership proof:

Exists: false
Hyper audit path: <TRUNCATED>
History audit path: <TRUNCATED>
CurrentVersion: 3
QueryVersion: 3
ActualVersion: 3
KeyDigest: 2d245d477b973c0895afc098b46762967f728e5aec8555d81ceaf1996d4c33e0
```

Important: Try restoring other backups and checking the membership of other events.

(repeat step 2 and 3 with different values)

6.14 Contributing

You can contribute in a few different ways:

- Submit issues through our issue tracker on Github.
- If you wish to make code changes, please check out above our guidelines about **Pull Requests** and the GitHub Forks/PullRequests model.

6.15 Pull requests

We have stablished a **work agreement** to provide a linear history, with at most one branch in parallel. We also require all commits in master to pass the tests.

For that to happen, this steps will enable you to get your pull request ready for being merged into the **master branch**. TL;DR: Always rebase to master before attempting to merge into master.

```
# download repo
git clone git@github.com:bbva/qed.git

# enter project dir
cd qed

# create your fork
hub fork

# create a to branch to hack on
git checkout -b my-cool-feature-branch

# ... do some groovy changes
git commit -am 'some explanatory although a bit cryptic msg ;-P'

# ensure your changes are in your github fork
git push my-user my-cool-feature-branch

# create PR
hub pull-request --base bbva:master

# wait for feedback (possibly master will advance in the meantime)
git commit ...
git commit ...
git commit ...
git push ...

# once it's approved and ready to merge, rebase to master and resolve all␣
↪conflicts.
git fetch origin master
git rebase origin/master

# push rebased branch to your fork (the PR will be updated automatically)
git push --force-with-lease my-user my-cool-feature-branch

# check again that tests are ok, and then merge (this step can only be␣
↪performed by developers with write access to the repo)
hub merge https://github.com/bbva/pr/pull/42
```

6.16 Github related projects

- Balloon
- GoSMT
- Trillian
- Continusec

6.17 Related papers

- https://github.com/google/trillian/blob/master/docs/VerifiableDataStructures.pdf
- http://tamperevident.cs.rice.edu/papers/paper-treehist.pdf
- http://kau.diva-portal.org/smash/get/diva2:936353/FULLTEXT01.pdf
- http://www.links.org/files/sunlight.html
- http://www.links.org/files/RevocationTransparency.pdf
- https://eprint.iacr.org/2015/007.pdf
- https://eprint.iacr.org/2016/683.pdf

6.18 Indices and tables

- genindex
- modindex
- search