Politecnico di Milano
School of Industrial and Information Engineering
Master of Science in Mathematical Engineering

Blockchain Notarization: extensions to the OpenTimestamps protocol

Supervisors: Prof. Daniele Marazzina
Prof. Ferdinando M. Ametrano

Master thesis by:
Andrea Brandoli
ID: 842282

Academic year 2018-2019

Contents

List of Tables **iv**

List of Figures **v**

List of Algorithms **vi**

Abstract **vii**

Acknowledgements **viii**

1 Introduction **1**
1.1 Thesis Structure 1

2 Distributed Consensus **3**
2.1 Distributed Systems 3
2.2 The Consensus Problem 4
2.2.1 The Byzantine Generals' Problem 6
2.2.2 Byzantine Agreement 7
2.2.3 Impossibility Result 10

3 Nakamoto Consensus in Bitcoin **12**
3.1 Eventual Consistency 12
3.2 Bitcoin Design 14
3.2.1 Constructing a Decentralized Digital Currency 14
3.2.2 Transactions and Blockchain Structure 16
3.2.3 Hash Functions 17
3.2.4 Mining & Proof of Work 20

4 Blockchain Notarization with OpenTimestamps **24**
4.1 Blockchain Timestamping 25
4.2 OpenTimestamps 29
4.2.1 Solving Scalability Problem 29

4.3 OpenTimestamps Python Client 32
4.4 OpenTimestamps Web Interface 36

5 Blockchain Notarization: A Practical Use Case 38
5.1 Executive Summary 38
5.2 Architecture of the Solution 39
5.3 Technical Details 41
5.3.1 Client Side 41
5.3.2 Server Side 42
5.3.3 Extensions to the protocol 42

6 Conclusions 44

A Bitcoin from the command line 45
A.1 Running a Bitcoin Core node 45
A.2 Bitcoin-cli: command line interface 47

List of Tables

List of Figures

2.1 Conquest of Constantinople 6
2.2 Lieutenant 2 is a traitor 11
2.3 Commander is a traitor 11

3.1 Illustration of the double spending problem 15
3.2 Blockchain as a hash pointer linked list 17

4.1 An example of a merkle tree of transactions 28
4.2 Scalability solution with *OpenTimestamps*. 30
4.3 Stamping 36
4.4 Verifying 37

5.1 Architecture of the timestamping process 40
5.2 Architecture of the verification process 40

List of Algorithms

2.1 King Algorithm (for $f < n/3$) 8
2.2 Asynchronous Byzantine Agreement (Ben-Or, for $f < n/9$) . . 10
3.1 Bitcoin network behaviour 21
4.1 Cooperation between aggregation and calendar servers 31

Abstract

Acknowledgements

Chapter 1

Introduction

1.1 Thesis Structure

In Chapter 2 we will start presenting the consensus problem in distributed systems: first we will define what a distributed system is and the main reasons in adopting it (Section 2.1). Then we will get into the consensus problem (Section 2.2), in particular in the case of an asynchronous network in presence of byzantine (malicious) nodes (Sections 2.2.1 and 2.2.2), also showing a famous impossibility result (Section 2.2.3) in that specific case.

Chapter 3 will delve into the Bitcoin's novel consensus mechanism. We initially think of Bitcoin as an example of eventual consistency, to focus the Nakamoto's clever intuition of relaxing some properties in the definition of consensus to hold only probabilistically (Section 3.1). Then we will give an overview of the main building blocks of the Bitcoin protocol (Section 3.2), with a particular interest for the underlying blockchain as a tamper-evident append-only log (Section 3.2.2). All this effort to completely understand how Nakamoto practically solves the consensus problem in Bitcoin and why this notorious blockchain is immutable.

The first two Chapters set the basis for the core of this thesis work, that is one of the few real non-monetary applications of the blockchain technology: notarization of digital documents, presented in Chapter 4. More specifically we will define what a timestamp is and how the timestamping procedure works (Section 4.1). We will also introduce *OpenTimestamps*, an open-source project that aims to be a standard for blockchain timestamping (Section 4.2), providing a step-by-step tutorial of the usage of its main releases (Sections 4.3 and 4.4).

Finally, in Chapter 5 we will present a technical report of a practical use case

of blockchain notarization, where the underlying *OpenTimestamps* protocol is extended and modified in the basis of need.

In Chapter 6 we will draw the conclusions of this thesis work.

Chapter 2

Distributed Consensus

2.1 Distributed Systems

With the persistent expansion of technology in the digital age we are going through, distributed systems are becoming more and more widespread.

On one side big companies operate on a global scale with thousands of machines deployed all over the world, big data are stored in various data centers and computational power is shared on multi-core processors or computing clusters.

On the other side, every day thousands of people withdraw money in a ATM point, a perfect example of distributed system. But simply, just think about a modern smartphone: it can share multiple data on the cloud and contain multiple co-working processing devices.

However, there is no a unique formal definition of distributed system. The one that fits better our interest is:

Definition 2.1.1. (distributed system)[1]. *A distributed system is a system whose components are located on different networked computers, which communicate and coordinate their actions by passing messages to one another. The components interact with one another in order to achieve a common goal. Three significant characteristics of distributed systems are: concurrency of components, lack of a global clock, and independent failure of components.*

Therefore, the main reasons for using distributed systems are:

[1] `https://en.wikipedia.org/wiki/Distributed_computing`

- Scalability: Distributed systems should be scalable with respect to geography, administration or size.
- Performance: Compared to other models, distributed models are expected to give a higher boost to performance.
- Fault Tolerance: A cluster of multiple machines is inherently more fault-tolerant than a single source of failure.
- Reliability: Data is replicated on different machines in order to prevent loss.
- Availability: Sharing data on different machines minimize latency and provide faster access.

However, beautiful features like these often have a downside, bringing to light some challenging problems:

- Security: Especially when networks are public.
- Coordination[2]: In public distributed systems coordination problems are prevalent if proper policies or protocols are not in place; there could be byzantine[3] agents.

Thus, it can be easily inferred that one of the main challenges of distributed systems is to achieve overall system reliability in the presence of a number of faulty processes. Examples of applications of consensus include whether to commit a transaction to a database, agreeing on the identity of a leader, state machine replication, clock synchronization, PageRank, load balancing and many others. Let us get in a detailed study of what consensus is and how consensus can be achieved in a distributed system.

2.2 The Consensus Problem[4]

Networks are composed by many agents, called *nodes*. Think of a computer network, nodes are either *honest*, executing programs faithfully, or *byzantine*

[2]In the fin-tech industry, coordination problems come with various names: consistency, agreement, consensus or blockchain.

[3]As specified in Definition 2.2.2.

[4]Definitions and Algorithms presented in this section are taken from [20], a book in which the author deeply delves into the problem of achieving consensus in distributed systems.

[13], exhibiting arbitrary behavior. They can also be *correct* and *faulty*, but not as alternatives to honest and byzantine. A correct node is an honest node that always eventually makes progress. A faulty node is a byzantine node or an honest node that has crashed or will eventually crash. Notice that the terms honest and byzantine are mutually exclusive, as are correct and faulty. However, a node can be both honest and faulty.

Definition 2.2.1. (node). *A single actor in the system is called node. In a computer network the computers are the nodes, in the classical client-server model both the server and the client are nodes, and so on.*

Definition 2.2.2. (byzantine node)[5]. *A node which can have arbitrary behaviour is called byzantine. This includes "anything imaginable", e.g., not sending any massage at all, or sending different and wrong messages to different neighbors, or lying about the input value.*

We assume that each pair of nodes is connected by a link, which is a bi-directional reliable virtual circuit and therefore messages sent on this link are delivered, eventually, and in the order in which they were sent (i.e., a sender keeps re-transmitting a message until it receives an acknowledgment or crashes). A receiver can tell who sent a message, thus a message cannot be forged. Standing the above consideration, a message sent by a byzantine node is indistinguishable from a message sent by an honest node.

In the consensus problem nodes run *actors*, that are either *proposers*, each of which proposes a *proposal*, or *deciders*, each of which *decides* one of the proposals. Assuming there exists at least one correct proposer (i.e., a proposer on a correct node), the goal of a consensus protocol is to ensure each correct decider decides the same proposal, even in the face of faulty proposers. A node may run both a proposer and a decider, in practice a proposer often would like to learn the outcome of the agreement.

Definition 2.2.3. (consensus). *There are n nodes, of which at most f might crash, i.e., at least $n - f$ nodes are correct. Node i starts with a proposed value v_i. The nodes must decide for one of those values, satisfying the following properties:*

- *Agreement: All correct nodes decide for the same proposal.*

[5]Before the term *byzantine* was coined, the terms Albanian Generals or Chinese Generals were used in order to describe malicious behavior. When the involved researchers met people from these countries they moved, for obvious reasons, to the historic term byzantine.

- *Termination: All correct nodes terminate in finite time.*
- *Validity: The decision value must be the proposal of some proposer.*

What is really interesting is to learn how to achieve distributed consensus in an asynchronous network in presence of byzantine nodes. Starting with the historical formulation of the problem and a first example of solved consensus in a model that is synchronous, we will finally face a famous impossibility result in the asynchronous case.

2.2.1 The Byzantine Generals' Problem

Distributed computer systems that want to be reliable must handle faulty components that give conflicting information to different parts of the system.

Figure 2.1: Conquest of Constantinople

Source: `https://medium.com/all-things-ledger/the-byzantine-generals-problem-168553f31480`

The "Byzantine Generals' Problem" is the abstraction of the aforementioned situation. As in Figure 2.1, we imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals communicate with one another as well as with all

lieutenants only by messenger. After observing the enemy, they must decide upon a common plan of action: the exact time to attack all at once or, if faced by fierce resistance, the time to retreat all at once. The army cannot hold on forever, if the attack or retreat is without full strength then brutal defeat is the only possible outcome. However, some of the generals may be traitors, trying to prevent loyal generals from reaching agreement.

For simplicity, we can restrict ourselves to the case of a commanding general sending an order to his lieutenants, obtaining the following problem.

Definition 2.2.4. (byzantine generals' problem). *A commanding general must send an order to his $n-1$ lieutenant generals such that the following conditions*[6] *are satisfied:*

1. *All loyal lieutenants obey the same order.*
2. *If the commanding general is loyal, then every loyal lieutenant obeys the order he sends.*

2.2.2 Byzantine Agreement

Achieving consensus (as in Definition 2.2.3) in a system with byzantine nodes is hard stuff. A careful reader immediately realizes that fulfilling agreement and termination is straight-forward, but what about validity? Reminding that a byzantine node can be malevolent lying about its input value, we must specify different types of validity:

Definition 2.2.5. (any-input validity)[7]. *The decision value must be the proposed value of any node.*

As we can see, this definition does not still make sense in presence of byzantine nodes; we would wish for a differentiation between byzantine and correct inputs.

Definition 2.2.6. (correct-input validity). *The decision value must be the proposed value of a correct node.*

Achieving this particular validity definition is not so simple, as byzantine nodes following correctly a specified protocol but lying about theirs input values are indistinguishable from correct nodes. An alternative could be:

[6] Conditions 1 and 2 are called the *interactive consistency* conditions

[7] This is the validity definition we implicitly used for consensus, in Definition 2.2.3

Definition 2.2.7. (all-same validity). *If all correct nodes start with the same proposed value v, the decision value must be v.*

Here, if the decision values are binary, then correct-input validity is induced by all-same validity. Else, if the input values are not binary, all-same validity is not useful anymore.

Now that we have clear in mind what are the ingredients for the consensus recipe, a question bales to us: which are the algorithms that solve byzantine agreement? What type of validity they fulfill? The King algorithm is one of the best examples, but we have to restrict ourselves to the so-called synchronous model.

Definition 2.2.8. (synchronous model). *In the synchronous model, nodes operate in synchronous rounds. In each round, each node may send a message to the other nodes, receive the message sent by the other nodes, and do some local computation.*

Algorithm 2.1 King Algorithm (for $f < n/3$)

```
1:  x = my input value
2:  for phase = 1 to f + 1 do
        Round 1
3:      Broadcast value(x)
        Round 2
4:      if some value(y) at least n − f times then
5:          Broadcast propose(y)
6:      end if
7:      if some propose(z) received more than f times then
8:          x = z
9:      end if
        Round 3
10:     Let node v_i be the predefined king of this phase i
11:     The king v_i broadcasts its current value w
12:     if received strictly less than n − f propose(x) then
13:         x = w
14:     end if
15: end for
```

To be rigorous, we must state some useful lemmas in order to prove that Algorithm 2.1 solves byzantine agreement.

Lemma 2.2.1. *Algorithm 2.1 fulfills the all-same validity.*

Lemma 2.2.2. *There is at least one phase with a correct king.*

Lemma 2.2.3. *After a round with a correct king, the correct nodes will not change their values v anymore, if $n > 3f$.*

Theorem 2.2.1. *Algorithm 2.1 solves byzantine agreement.*

Proof. The king algorithm reaches agreement as either all correct nodes start with the same value, or they agree on the same value latest after the phase where a correct node was king according to Lemmas 2.2.2 and 2.2.3. Because of Lemma 2.2.1 we know that they will stick with this value. Termination is guaranteed after $3(f + 1)$ rounds, and all-same validity is proved in Lemma 2.2.1. □

However, this is not the end of the story about distributed consensus. In order to dig into the hearth of this work we must introduce consensus results in the asynchronous model.

Definition 2.2.9. (asynchronous model). *In the asynchronous model, algorithms are event based ("upon receiving message ..., do ..."). Nodes do not have access to a synchronized wall-clock. A message sent from one node to another will arrive in a finite but unbounded time.*

Definition 2.2.10. (asynchronous runtime). *For algorithms in the asynchronous model, the runtime is the number of time units from the start of the execution to its completion in the worst case (every legal input, every execution scenario), assuming that each message has a delay of at most one time unit.*

We will now present a famous algorithm (Algorithm 2.2) by Ben-Or [9], which tries to solve asynchronous byzantine agreement.

Unfortunately, Algorithm 2.2 is just a proof of concept that asynchronous byzantine agreement can be achieved, but practically it is unfeasible due to its exponential runtime.

Algorithm 2.2 Asynchronous Byzantine Agreement (Ben-Or, for $f < n/9$)

1: $x_i \in \{0, 1\}$ ▷ input bit
2: $r = 1$ ▷ round
3: decided = false
4: Broadcast $propose(x_i, r)$
5: **repeat**
6: Wait until $n - f$ *propose* messages of current round r arrived
7: **if** at least $n - 2f$ *propose* messages contain the same value x **then**
8: $x_i = x$, decided = true
9: **else if** at least $n - 4f$ *propose* messages contain the same value x **then**
10: $x_i = x$
11: **else**
12: choose x_i randomly, with $Pr[x_i = 0] = Pr[x_i = 1] = 1/2$
13: **end if**
14: $r = r + 1$
15: Broadcast $propose(x_i, r)$
16: **until** decided (see Line 8)
17: decision = x_i

2.2.3 Impossibility Result

A solution to the Byzantine Generals' Problem 2.2.4 may seem a piece of cake. That's not true at all. Its difficulty arises from the fact that if the generals can send only oral messages, then no solution will work unless more than two-thirds of the generals are loyal. An oral message is one whose contents are completely under the control of the sender, so a traitorous sender can transmit any possible message. Such a message corresponds to the type of message that computers normally send to one another.

Let us now show that, if only oral messages are allowed, there is no solution for three generals with a single traitor. For simplicity, the only possible messages that generals can send are "attack" or "retreat". This restriction opens two possible scenarios. In the first scenario, shown in Figure 2.2, a loyal commander sends an "attack" order to his lieutenants, but lieutenant 2 is malevolent and report "retreat" to lieutenant 1. In this situation, in order to satisfy Condition 2 of problem 2.2.4, lieutenant 1 must obey the order to attack. In the second scenario, shown in Figure 2.3, the commander is a traitor and sends an "attack" command to lieutenant 1, and a "retreat" order to lieutenant 2. But, lieutenant 1 does not know who is the traitor and

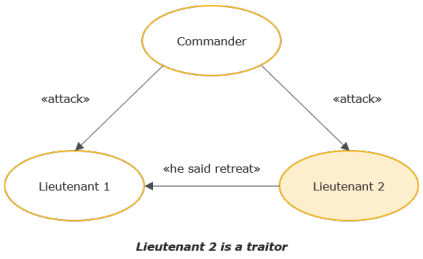

Figure 2.2: Lieutenant 2 is a traitor

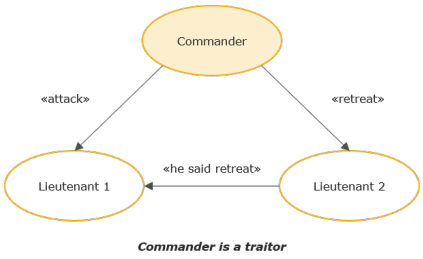

Figure 2.3: Commander is a traitor

cannot tell the order the commander sent to lieutenant 2. Thus, lieutenant 1 cannot distinguish in which scenario he belongs, so he always obey to the "attack" order. With a similar argument, if lieutenant 2 receive a "retreat" order from the commander, he must follow him even if lieutenant 1 reports him to attack. This violates Condition 1 of problem 2.2.4.

A demanding reader could comply about this nonrigorous result. A famous result[8] of Fisher, Lynch and Paterson [11] formally proves impossibility of distributed consensus with one faulty process.

Due to this proven impossibility result, how Satoshi Nakamoto reaches consensus in Bitcoin, a decentralized, distributed, peer-to-peer network? In the next chapter we will delve into this argument.

[8]This result was awarded the 2001 PODC Influential Paper Award (now called Dijkstra Prize).

Chapter 3

Nakamoto Consensus in Bitcoin

Bitcoin is a distributed, decentralized, peer-to-peer electronic payment system based on cryptographic proof instead of trust, allowing transactions between two counterparts without the need for a trusted third party. However, in late 2008, when the white paper was published by his author Satoshi Nakamoto[1] [14], it lacked a formalization of the protocol and of the guarantees it claimed to provide.

This chapter delves into the core innovation behind Bitcoin, i.e. *Nakamoto consensus*, term that is commonly used to refer to Bitcoin's novel consensus mechanism, which allows mutually distrusting pseudonymous[2] identities to reach eventual agreement.

3.1 Eventual Consistency

By its very nature, the Bitcoin network is subject to a type of failure called *network partition*, where a network splits into at least two parts that cannot communicate with each other, often due to software bugs, incompatible protocol versions, or simply network disconnections. Thus, Bitcoin is inherently

[1]The name Satoshi Nakamoto is the pseudonym adopted by the creator of Bitcoin. While his identity remains a mystery, some information is known. He registered the domain bitcoin.org in August 2008 and in October 2008 he publicly released the famous white paper. In the Bitcoin's early days he participated extensively in forums and mailing lists maintaining the souce code. During the next two years other contributors slowly taken over the project maintenance and he stopped communicating, leaving a veil of mystery.

[2]Agents in the bitcoin network play under a pseudonym, that is the *address* to which they receive transactions. As specified in the original white paper [14], Bitcoin users are recommended to use a new address for each transaction in order to avoid that many transactions are connected to a common owner, leading anonymity.

characterized by a trade-off between *consistency*, *availability* and *partition tolerance*. Let us be more precise.

Definition 3.1.1. (consistency). *All nodes in the system agree on the current state of the system.*

Definition 3.1.2. (availability). *The system is operational and instantly processing incoming requests.*

Definition 3.1.3. (partition tolerance). *Partition tolerance is the ability of a distributed system to continue operating correctly even in the presence of a network partition.*

In practice, only two of these properties can be reached simultaneously; a theorem by Brewer proves this result.

Theorem 3.1.1. (CAP theorem). *It is impossible for a distributed system to simultaneously provide consistency, availability and partition tolerance. A distributed system can satisfy any two of these but not all three.*

Proof. Let us assume two nodes that share some state. The nodes belongs to different partitions, so they cannot communicate each other. Assume a request wants to update the state and contacts one of the two nodes. The node may either: 1) update its local state, resulting in a situation of inconsistent states, or 2) not update, resulting in a system no longer available for updates. □

Recalling the Definition 2.2.3 of consensus and the properties it must satisfy, a clever intuition of Nakamoto is to weaken the agreement property to hold probabilistically and not deterministically, in order to deal with an asynchronous network. In particular the aforementioned trade-off is in advantage of partition tolerance rather than consistency. In fact, state changes of the underlying transaction ledger (i.e. the blockchain), are rendered probabilistic and the decision on a specific value of the state reaches $Pr(1)$ when $\lim_{r\to\infty}$, where r is the number of rounds in the consensus protocol.

Therefore, we can look at Bitcoin as an example of *eventual consistency*.

Definition 3.1.4. (eventual consistency) *If no new updates to the shared state are issued, then eventually the system is in a quiescent state, i.e., no more messages need to be exchanged between nodes, and the shared state is consistent.*

Now that we have clear in mind the consensus problem from a computer science perspective, we must move forward and study how practically Nakamoto consensus works. Starting from a brief summary of some basic Bitcoin mechanics, we will finally show how cleverly Nakamoto solves the consensus problem in Bitcoin, going through cryptography tools and game theory aspects.

3.2 Bitcoin Design

We already defined Bitcoin as a distributed, decentralized, peer-to-peer electronic payment system, but we lacked about defining the underlying digital asset, i.e. *bitcoin*, the challenges in building it, and also the fundamental bricks of the Bitcoin wall. Combining all these aspects is crucial to fully understand Nakamoto consensus.

3.2.1 Constructing a Decentralized Digital Currency

Ownership is the first challenge of constructing a virtual currency. Imagine units of fiat currencies, their ownership depends on their form. If these units are in the form of paper notes or metal coins, ownership is simply determined by physical possession. Else, if these units are digitally stored in a bank account, ownership is determined by possession of the credentials to spend them. But what about ownership for a decentralized virtual currency? Who maintains the ledger of credentials? How can we guarantee the integrity of that ledger? In a network like Bitcoin nodes are free to leave at any time, thus a single source of trust (a node or a group of them) cannot be given this responsibility, even if one-way functions protects credentials. A solution could be that all nodes maintain a copy of that ledger, but what about validation? Can I feasible query all the nodes? Integrity is also crucial: usually digital signatures ensure it, but reminds the nodes could act maliciously and so, can I trust the signer?

Another big challenge is the so called *double spending* problem. This refers to the fact that the owner of some units of digital currency could spend them more than once. With fiat currencies, granted by a central authority, this problem cannot occur. In fact the bank database is the single source of truth with regard to the deposited amount of currency. Thus, spending with f.e. an online transfer, would immediately result in the rebalancing of that account to reflect the spend. However, in a peer-to-peer decentralized network some

malicious agent could try to perform a double spend taking advantage of the delay in updating the shared ledger. An example will explain better.

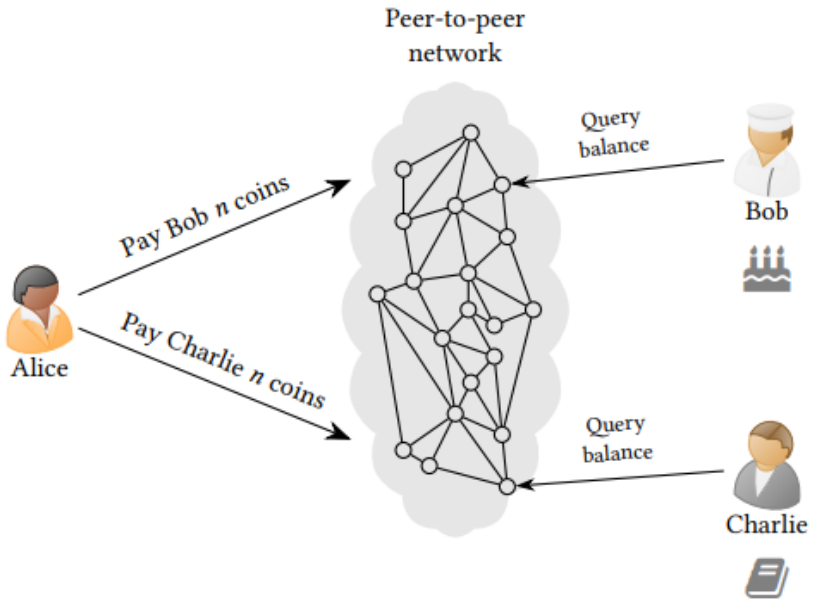

Figure 3.1: Illustration of the double spending problem

Consider the scenario described in Figure 3.1 where Alice tries to perform a double spend using some digital coins in her wallet. She would purchase a cake from Bob and a book from Charlie spending the same n coins. For simplicity, let us assume that the two items cost the same, precisely n units of currency. Thus, in order to finalize the purchase, both Bob and Charlie will provide Alice with their wallet addresses. Then Alice creates two different transactions, both spending the same n coins, one paying Bob for the cake and the other paying Charlie for the book. The network will only accept one of them because the two transactions conflict. But there are several nodes in the network, some of which can be made to accept one of the transactions and the rest to accept the other. So Alice can transmit the transaction paying Bob to the portion of the network which Bob is connected to and the same argument holds for Charlie's transaction. Therefore, when Bob and Charlie query the balances in their private wallets they both will find a valid payment. If they provide Alice with their goods before being aware that the two transactions are conflicting, the result is that Alice has successfully performed a double spend.

Nakamoto solves brilliantly the aforementioned challenges in Bitcoin combining cryptography and social incentive engineering. But first we need a brief summary of the mechanics of the protocol.

3.2.2 Transactions and Blockchain Structure

In Bitcoin, a coin or a *bitcoin* is defined as a chain of digital signatures. Coins cannot be combined, subdivided or transferred, they can only be entirely consumed as transaction inputs (TxIn) in order to create new output coins (TxOut). Thus, a transaction output can only be in two states, *spent* or *unspent*. The set of current unspent coins takes the name of UTXO (Unspent Transaction Outputs). So, a *transaction* assumes the role of the fundamental unit of state in the Bitcoin protocol.

More precisely, each Tx[3] output consists of an amount of bitcoins and a cryptographic puzzle, the so-called *locking script*, which determines the spending conditions for that amount[4]. An example of such a puzzle could be a request for a digital signature which can be validated with a public key. On the other hand, a Tx input is characterized by a pointer referencing the UTXO that it consumes and an *unlocking script*, the solution to the aforementioned locking script, which proves that funds in the considered transaction can be correctly spent.

All the transactions, starting from the time the protocol was launched[5], are securely recorded in a distributed ledger, shaped as a sequence of blocks, the *blockchain*. In a technical cryptographic perspective, the *blockchain* is a hash pointer linked list, whose aim is to be a *tamper-evident* log. That is, a log data structure that stores some data (mainly transactions) allowing only to append new data onto the end of the log but, if somebody alters some earlier data then it would be easily detected. Figure 3.2 provides a graphic idea of the structure.

[3]It's a common practice in Bitcoin community to abbreviate the word *transaction* as Tx or sometimes TX.

[4]Bitcoin uses a script system for transactions, called *Script*. It is a simple language, Forth-like, processed from left to right using a stack. In addition, it is designed such that it guarantees all scripts will execute in a limited amount of time (i.e. it is not Touring-Complete).

[5]The first block of the chain, called the *genesis* block, was generated in January 2009

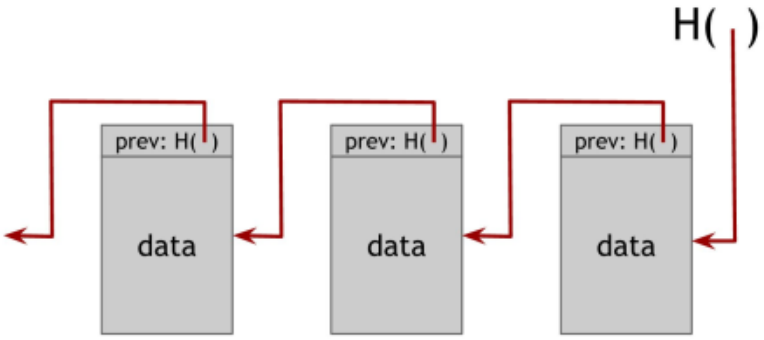

Figure 3.2: Blockchain as a hash pointer linked list

Source: [15]

The aforementioned *tamper-evidence* property derives from cryptography, so we must introduce some primitives. But first a bit of history. The ideas behind the blockchain trace back to a paper by Haber and Stornetta in 1991 [12]. They proposed a method to *timestamp* digital documents, that is a method to give an idea of when a document came into exixtance. To achieve this goal, a timestamping server receives client documents to timestamp. Then it signs the documents (one by one) together with the current time, and a special type of pointer that links to a piece of data instead of a location, pointing to the previous document. Thus, if some data is tampered, that pointer turns to be invalid. Finally the timestamping server issues a certificate containing such information. Therefore, it is clear that each document's certificate ensures the integrity of the previous document.

3.2.3 Hash Functions

The more the world becomes digital, the more we need information security. Cryptography mixes advanced mathematical and computer science techniques to provide confidentiality, data integrity, authentication and non-repudiation.

A *hash function* is a particular mathematical function that maps an arbitrary length input message into a short, fixed-length bit string, usually called *digest* or *hash value*. Hash functions can be seen as a digital fingerprint of a message, i.e. a unique representation of a message.

Definition 3.2.1. *A function* $h : \{0,1\}^* \to \{0,1\}^n$ *is a hash function if it is computable in polynomial time in the length of the input.*

This definition lacks of security arguments. There are three main properties that hash functions need to possess in order to provide security:

Definition 3.2.2. *Let h be a hash function, the following properties may hold:*

- *preimage resistance (one-wayness): given $h(x)$ it is not feasible to compute x;*
- *second-preimage resistance (weak collision resistance): given x it is not feasible to compute y s.t. $x \neq y$, $h(x) = h(y)$;*
- *collision resistance (strong collision resistance): it is not feasible to find x, y s.t. $x \neq y$, $h(x) = h(y)$.*

A hash function that fulfills the preimage resistance property *hides* the input, so one cannot derive a matching message from a hash value. A second-preimage resistant hash function prevents that, given a message, one can find another message that produces the same output once hashed. Unfortunately such a property cannot exist in theory, due to the so-called *pigeonhole principle*[6]. Suppose to be the owner of 100 pigeons but in your pigeon loop are only 99 holes, at least one pigeonhole will be occupied by 2 birds. The same argument holds for hash functions. Since the output of a hash function has a fixed bit length, say n bits, there exist only 2^n possible output values. However the length of input values is arbitrary, resulting in an infinite number of possible inputs. Thus, weak collision can occur in theory. Fortunately, given today's computers, an output length of $n = 80$ bits is sufficient to resist an analytical attack, that is, given x and $h(x)$ one wants to construct y such that $h(y) = h(x)$. To sum up, collisions do exist, but they are *unfeasible*. Finally, the strong collision resistance property refers to the problem of finding two messages that generate the same hash value. With an output length of $n = 80$ bits as before, producing 2^{80} possible values, one may think that finding two inputs generating the same output would require around $\frac{2^{80}}{2}$, a half of the possible values. That's completely wrong, due to the *birthday paradox*, a powerful cryptographic tool which deserves a formal definition.

Definition 3.2.3. (birthday paradox). *The birthday paradox concerns the probability that, given a set of n randomly chosen people, some pair of them will have the same birthday.*

[6]Formally this principle takes the name of *Dirichlet's drawer principle*, but it is commonly referred to as the *pigeonhole principle*.

The correct approach to solve this problem is to compute the probability of two people *not* having the same birthday:

$$P(\text{2 people not colliding}) = \left(1 - \frac{1}{365}\right)$$

If a third person is joining:

$$P(\text{3 people not colliding}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right)$$

Hence, the probability for t people having no birthday collision is given by:

$$P(t \text{ people not colliding}) = \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{t-1}{365}\right)$$

Then, making some simple calculations, it turns out that it only requires 23 people to have a probability of about 0.5 for a birthday collision:

$$\begin{aligned} P(\text{at least one collision}) &= 1 - P(\text{no collision}) \\ &= \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{23-1}{365}\right) \\ &= 0.507 \approx 50\% \end{aligned}$$

The search of a collision for a hash function is exactly the same problem as the birthday problem, obviously with different parameters. Again, let n be the output length in bits, thus resulting in 2^n possible output values. Here, the probability of no collisions among t hash values is:

$$\begin{aligned} P(\text{no collision}) &= \left(1 - \frac{1}{2^n}\right) \cdot \left(1 - \frac{2}{2^n}\right) \cdots \left(1 - \frac{t-1}{2^n}\right) \\ &= \prod_{i=1}^{t-1} \left(1 - \frac{i}{2^n}\right) \\ &\approx \prod_{i=1}^{t-1} e^{-\frac{i}{2^n}} \\ &\approx e^{-\frac{t(t-1)}{2 \cdot 2^n}} \end{aligned}$$

Remind that we want to find how many messages are required to find a collision. Let λ be the probability of at least one collision:

$$\lambda \approx 1 - e^{-\frac{t(t-1)}{2^{n+1}}}$$

Solving for t and observing that, since in practice $t >> 1$, it holds that $t^2 \approx t(t-1)$ we obtain:

$$t \approx 2^{(n+1)/2} \sqrt{\ln\left(\frac{1}{1-\lambda}\right)} \tag{3.1}$$

The main takeaway of the result obtained in Equation 3.1 is that the number of messages needed to find a collision of hash values is proportional to the square root of the number of possible output values.

Example 3.2.1. *SHA256*[7] *(Secure Hash Algorithm) is the hash function the Bitcoin protocol is built upon. It produces a 256 bit output digest and, at the moment, it is considered to satisfy all the properties described in Definition 3.2.2. Moreover, SHA256 could be thought as a random oracle, a fixed input will provide always the same output, since hash functions are deterministic, but the outputs corresponding to new inputs are indistinguishable from a uniform distribution. Thus, a small perturbation in the input string will result in a completely different digest.*

```
SHA256("I love you")  = c33084feaa65adbbbebd0c9bf292a26ffc6
                        dea97b170d88e501ab4865591aafd
SHA256("I love you!") = c22fe0282c7b45b0926fb3e33efd375a5e1
                        95c4c838c96075e3877b2ac2b7911
```

3.2.4 Mining & Proof of Work

The Bitcoin blockchain is a public distributed ledger of transactions. Being public means that anyone should be allowed to add transactions to the blockchain. However, we already saw that in a peer-to-peer asynchronous network like this, some nodes are byzantine and could be malevolent, trying to perform doublespends, which can result in an inconsistent state since the validity of transactions depends on the order in which they arrive, not the same for all nodes. Thus, if doublespends are not resolved, the shared state diverges. Hence, we need a mechanism to resolve conflicting transactions, the so-called *proof-of-work*[8], in order to append only valid blocks to the blockchain.

[7] SHA256 belongs to the SHA2 family. Nowadays SHA3 family is available, even more secure.

[8] The idea of *proof-of-work* is taken from Adam Back's Hashcash [7], a scheme to combat email spam proposed in 1997.

Definition 3.2.4. (proof-of-work). *The proof-of-work (pow) is a mechanism allowing a party to prove to another party that a certain amount of computational power has been used for a period of time. A function* $F_d(c, x) \rightarrow \{true, false\}$*, where the difficulty* d *is a positive number, while the challenge* c *and the nonce* x *are usually strings of bits, is a pow function if it satisfies the following properties:*

1. *if* d*,* c *and* x *are given,* $F_d(c, x)$ *is fast to compute.*
2. *For fixed* c *and* d*, finding* x *s.t.* $F_d(c, x) = true$ *is computationally hard but feasible. In addition, the difficulty* d *serves to adjust the time to find that* x*.*

However, the proof-of-work mechanism by itself does not fully solve the double-spending problem. But, if correctly used, it makes the blockchain tamper-resistant, thus extremely hard and expensive to rewrite. We will see how the proof-of-work is shaped in Bitcoin just in a while, inside the explanation of the network behaviour.

The clever intuition of Nakamoto in resolving doublespends is to incentivize network nodes to behave honestly, checking for and rejecting fraudolent transactions. The procedure by which new blocks are added to the blockchain while ensuring that they contain only valid transactions is called *mining*. In addition, this procedure controls the rate of generation of new bitcoins. Before delving into the details of the mining procedure, we have to show a summary algorithm of the Bitcoin network behaviour, presented in the original white paper by Nakamoto [14].

Algorithm 3.1 Bitcoin network behaviour

1: New transactions are broadcast to all nodes.
2: Each node collects new transactions into a block.
3: Each node works on finding a difficult proof-of-work for its block.
4: When a node finds a proof-of-work, it broadcasts the block to all nodes.
5: Nodes accept the block only if all transactions in it are valid and not already spent.
6: Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

Taking advantage of the main steps of the Bitcoin network behaviour described in Algorithm 3.1, we can proceed to define the mining procedure as a workflow:

1. People performing bitcoin transactions will broadcast them onto the network. Such transactions will held in the *mempool* (memory pool), that is a collection of pending transactions, before being included in a block. Each node of the network operate its own mempool, with its own unique size (typically MB).

2. Some special nodes, called *miners*, collect some of these transactions into a *candidate block*, which could be different for each miner, composed by a *block header* and a list of transactions. The first transaction in the block is called *coinbase transaction*. This special transaction has a dummy input, called *coinbase*, with no references to parent TxOut, and creates new bitcoins from nothing. In addition, each transaction includes a small *fee*, as the difference between inputs and outputs. Finally, the *block header* consists of the following fields:

   - A block version number.
   - The double SHA256 of the previous block header.
   - The *merkleroot*[9] of all transactions in that block.
   - The block generation time (Unix epoch time) according to the miner.
   - An encoded version (nBits) of the target threshold this block header hash must be less than or equal to.
   - The nonce used to effectively mine that block.

3. The sum of the transaction fees included in all regular transactions, plus the bitcoins generated in the coinbase transaction is called *block reward*[10] and it is collected by the block's miner, allowing it to mint new coins. These economic incentives make Bitcoin a very expensive network, but they trigger a virtuous cycle[11], encouraging nodes to behave honestly competing to append valid blocks to the blockchain, making it almost immutable. So, each miner will compete to find that specific *nonce* which verifies the Bitcoin proof-of-work. More precisely, recalling Definition 3.2.4:

[9] A *merkleroot* is the root of a *merkle tree* as presented in Definition 4.1.3.

[10] The number of bitcoins generated by the coinbase transaction is determined by a subsidy schedule that is part of the protocol. Started with 50 bitcoins for every block, it halves every 210000 blocks (about 4 years, considering an average of a block per 10 minutes). Nowadays it consists of 12.5 bitcoins. Due to this halving, the total amount of bitcoins never exceeds 21 million of units, making this asset scarce by nature.

[11] In an economic perspective, scarcity increases bitcoin market price.

Definition 3.2.5. (Bitcoin proof-of-work). *The Bitcoin proof-of-work consists of finding a special number x called nonce s.t. the double SHA256 of the candidate block header falls below a given threshold*[12]*:*

$$F_d(c, x) \rightarrow SHA256(SHA256(\underbrace{...\|prev\_block\_header\_hash\|...\|x}_{Candidate\ Block\ Header})) < \frac{2^{224}}{d}$$

4. The miner who first finds such a nonce will broadcast his own candidate block to the network. If all the transactions in that block pass the validity check, it is appended to the miner's local copy of the blockchain. Other nodes express their acceptance of the block by working on creating a new block using the hash of the accepted block as the previous hash. However, this is not the end of the story. It is possible that two miners mine their respective candidate blocks almost at the same time and, due to connection lag, they broadcast their blocks before being aware one of the other. This situation leads to a chain *fork*, that is two possible valid chains, at least for the moment. The solution to this type of problem is achieved by the *proof-of-work consensus*, i.e., in favor of the chain which required the most computational work[13] to produce.

We set the basis for the very hearth of this thesis work, that is *notarization* of digital documents with the Bitcoin blockchain. In Chapter 4 we will study what a *timestamp* on the blockchain is, introducing *OpenTimestamps*, an open-source project whose aim is to be a standard for blockchain notarization. Then, in Chapter 5 we will delve into the details of a project in which the author, in cooperation with DGI (Digital Gold Institute) and ANIA (Associazione Nazionale fra le Imprese Assicuratrici), built a fully operating timestamping service, modifying and extending the *OpenTimestamps* protocol on the basis of need.

[12]More precisely, the difficulty d is calculated as: $d = \frac{\text{difficulty\_1\_target}}{\text{current\_target}}$. Traditionally, the *difficulty_1_target* represents a hash where the leading 32 bits are zero and the rest are one, known as *pool difficulty*, or *pdiff*. Thus, the threshold value appears as $\frac{2^{224}}{d}$ and not $\frac{2^{256}}{d}$. In addition, the mining threshold is adaptively adjusted every 2016 blocks to achieve an average goal of a block 10 minutes (i.e. every 2 weeks).

[13]That is, with the majority of the network total hash rate working on it.

Chapter 4

Blockchain Notarization with OpenTimestamps

We devoted Chapter 2 and Chapter 3 in understanding how Bitcoin achieves distributed consensus, thus convincing ourselves that the relative blockchain is, in geek speak, *immutable*. We made this effort to lay the foundations for one of the most interesting non-monetary application[1] of the blockchain, that is the *notarization* of digital documents. According to National Notary Association[2]:

"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."

Traditionally, the notarization process is achieved by certification authorities (CA), like notary public or banks, because of their reliability. It is only thanks to the relationship of trust with social organizations if this task is possible. The majority of notary services are based on ledgers attesting non-repudiable and non-alterable transactions between two counterparts.

However, a central authority represents a single point of failure. Who maintains that ledger could act maliciously and easily tamper some data for his own interest. Hence the need to decentralize the source of trust and grant the

[1] Behind the hype for the blockchain in the current years as the definitive solution of all the world's problems, notarization of digital documents stands out as one of the few real applications of this technology.

[2] More details in the official website: `https://www.nationalnotary.org/knowledge-center/about-notaries/what-is-notarization`.

reliability of that ledger even if the security of the aforementioned central authority breaks from the inside. A careful reader will have already figured out that the blockchain is the technology that perfectly fits our interest, assuring tamper-resistance and non-repudiation of data written in the chain. Among them, permissionless (public) blockchain is more preferred, because a service built on top of a permissioned (private) blockchain represents a single point of failure, as for a central authority. So, it results that the Bitcoin blockchain is the right answer[3]. Unfortunately, at least for now, it cannot replace all the notary services, but it is perfect to give *proof-of-existance* of a document, a procedure usually called *timestamping*.

4.1 Blockchain Timestamping

A timestamp can be seen as a sequence of characters representing the time in which an event occurred. Hence, timestamping is an increasingly valuable complement to digital signing practices, enabling organizations to record when a digital item, such as a message, document, transaction or piece of software, was signed. In addition, the timing of a digital signature is critical in many cases, like lottery ticket issuance and some legal proceedings. Even when time is not intrinsic to the application, timestamping is helpful for record keeping and audit processes, because it proves whether the digital certificate was valid at the time it was used. More precisely:

Definition 4.1.1. (timestamp). *A timestamp is a proof that some data d existed prior to a certain time t.*

To create such proof, d has to cause an event that could not have been generated without the existence of d. Such event must be attested to time t and can be publicly observed. So a proof consists in the data d, the set of operations binding d to the time t, and the time attestation. However, we have to deal with the fact that digital documents can be easily falsified without leaving any trail of tamper evidence. Hence, a "good" timestamp proof must become invalid even if a single bit of the input string is modified. Once again cryptography comes in help and the creation procedure of a timestamp proof does not require to publish d on the blockchain, but only

[3]Regarding this, Italian law recognizes the validity of blockchain notarization and AGID will have to specify technical detail.
More info here: `https://www.agendadigitale.eu/documenti/blockchain-nel-ddl-semplificazioni-conseguenze-e-problemi-dellattuale-testo/`

a *commitment* to it, that is something caused by the input d or, in other words, something that follow such input in time. Let us be a little bit more precise:

Definition 4.1.2. (commitment operation). *A function $C : X \to Y$ is a commitment operation if given $x_1 \in X$ it is not feasible to compute $x_2 \in X$ s.t. $x_1 \neq x_2, C(x_1) = C(x_2)$.*

Recalling Definition 3.2.2, requiring a function to be a *commitment* operation is the same to require that function to fulfill *second-preimage resistance*, a property already seen in Paragraph 3.2.3 for hash functions. We will now provide some simple examples of commitment operations in order to understand better, the "*append*" and "*prepend*" operations.

Example 4.1.1. ("append"). *"hello"* $\xrightarrow{append(\text{"world"})}$ *"helloworld".*

Example 4.1.2. ("prepend"). *"world"* $\xrightarrow{prepend(\text{"hello"})}$ *"helloworld".*

It is clear that these commitment operations bind inputs to outputs, thus making impossible to tamper the inputs without any evidence in the outputs. However, any user of a timestamp service would have its privacy respected, thus not revealing the content of the data d, a property that neither "*append*" nor "*prepend*" are able to provide. In addition, the outputs of these specific operations are always bigger in size than the inputs, not a good feature if we want to record such commitments in the blockchain, having blocks strict size limits.

To address these issues, we need a tailored commitment operation that fits our case, that is the *hash function*. As we already saw in Paragraph 3.2.3, hash functions achieve specific properties:

- The same input data d will always yield the same hash value.
- The hash value is always of a fixed length[4], no matter what size of the input data d.
- Any change in the input data d, even a single bit, will have a totally different hash value and therefore can be detected.

[4]Accordingly to the Bitcoin protocol, we will use SHA256, resulting in a 256-bits output length.

- It is impossible to recover the original data d from the hash value, so it hides the input.

Now that we have seen all the properties a good commitment operation must achieve, what is left is to learn how blockchain timestamp effectively works.

Hash functions, like SHA256, are implemented as open algorithms, so any client can "hash" the document he wants to timestamp from any computer and using any programming languages, without the need to trust any third party for this task.

Once the hash value is computed, it can be associated to a particular Bitcoin transaction, called *null data transaction*. This specific type of transaction makes use of OP_RETURN, an opcode script introduced in the Bitcoin Core 0.9.0 release. It allows to add arbitrary data to a provably unspendable pubkey script that full nodes do not have to store in their UTXO dataset. This feature is good at preventing the UTXO dataset to bloat. From Bitcoin Core 0.11.0, *null data transactions* are relayed and mined with up to 80 bytes[5] in a single data push with the limitation of one null data output paying exactly 0 satoshis[6]:

```
Pubkey Script: OP_RETURN <0 to 80 bytes of data>
```

This *null data script* cannot be spent, so there is not an associated *unlocking script*. Then, the aforementioned transaction is broadcast to the network, waiting for a miner to add it in the transaction set of a mined block. We already saw that a block header contains a specific field called *merkleroot* when we talked about mining in Paragraph 3.2.4, substantially the root of a binary tree that summarize all the transactions in that block or, in other words, a *commitment* to them. We will use this tree structure, called *merkle tree*[7], in the next chapters, thus we will give here a brief explanation:

Definition 4.1.3. (merkle tree). *In cryptography and computer science, a binary hash tree or merkle tree is a tree in which every leaf node is labelled with the hash of a data block, and every non-leaf node is labelled with the cryptographic hash of the labels of its two child nodes.*

The simple way to fix in mind such a structure is by a figure. Antonopoulos in his book "Mastering Bitcoin" [6] did it for us.

[5]From version 0.9.x to 0.11.x of the Bitcoin Core, the size for pushing arbitrary data in the null data transaction was 40 bytes.

[6]Amounts in Bitcoin are expressed in *satoshi*, currently the smallest unit of the bitcoin currency recorded on the blockchain, i.e. 10^{-8} bitcoin.

[7]The name *merkle tree* comes from Ralph Merkle, who patented it in 1979.

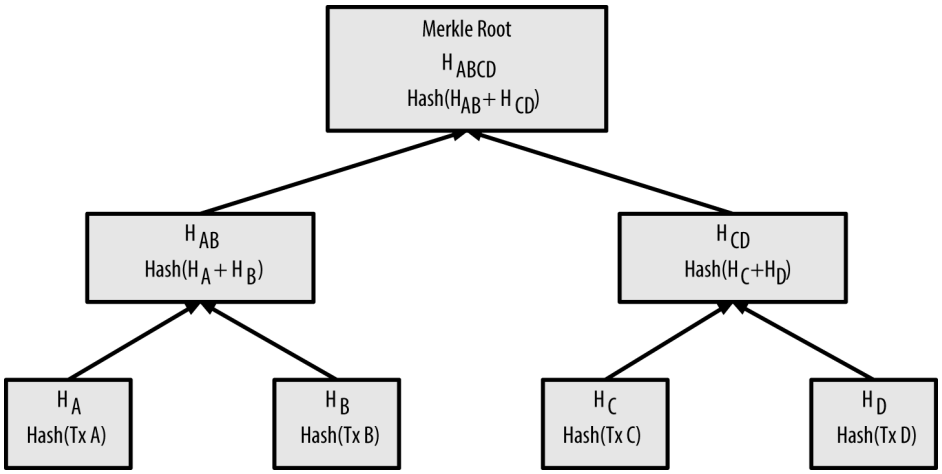

Figure 4.1: An example of a merkle tree of transactions

Source: `https://github.com/bitcoinbook/bitcoinbook/blob/develop/images/mbc2_0902.png`

A careful reader will already have understood that, in our case, data in Definition 4.1.3 are transactions and the hash function used is SHA256. Hence, for the properties of hash functions, the *merkleroot*, so the block header commits to all the transactions in a block, also to the specific one that in turn commits to the data to timestamp.

Finally, thanks to what we have learned in the previous chapters, we can think of the Bitcoin network as a decentralized, trustless, permissionless notary. Its attestations are stored in the blockchain, a widely published and immutable timestamps chain[8]. It is this immutability to provide timestamping, proving the data file existence at that moment in time in that specific status. Manca immagine timestamp.

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. In addition, it can be used along with timestamping prescription. However, this procedure has some downsides, being not efficient (it needs a transaction per timestamp) and lacking a proper standardization. But, in 2012 Peter Todd started working on an open-source

[8]We recall that a block header also contains a *time* field, a Unix epoch time when the miner started hashing the header (according to the miner). Its value must be strictly greater than the median time of the previous 11 blocks. Full nodes will reject blocks with headers more than two hours in the future according to their clock. This definition is taken from `https://bitcoin.org/en/developer-reference#block-headers`.

project[9], called *OpenTimestamps* [4], that aims to provide a standard format for blockchain timestamping, and efficiently resolves these downsides.

4.2 OpenTimestamps

OpenTimestamps defines a set of rules for conveniently creating provable timestamps and later independently verifying them. Currently this protocol fully supports Bitcoin blockchain timestamping, however it is flexible enough to support timestamping on other blockchains like Ethereum and Litecoin. Obviously such a proof is reliable as the chain committing to it, hence without a doubt the Bitcoin one is nowadays the best choice.

A timestamp proof made with *OpenTimestamps* consists in a list of commitment operations applied in sequence to the document, 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.

4.2.1 Solving Scalability Problem

As we mentioned before, *OpenTimestamps* proposes a solution to the scalability problem, allowing to efficiently aggregate possibly up to an infinite number of documents to timestamp in a single transaction, a feature non supported by most of the existing timestamping services. The trick is to compose those documents in a *merkle tree* and to push its *merkleroot* into a transaction, as the classic blockchain timestamping solutions did for a single hash value. Hence, each per-document proof is just the path up to the first merkle tree composed by documents' hashes, then up to the merkle tree of transactions, reaching the merkleroot in the block header which alone is a commitment to all the documents.

[9]The earliest Git commits date back to June 2012, as you can see here: `https://github.com/opentimestamps/opentimestamps-server/commit/6e2519d0bb8af2c6ec006e7849d626b20af99a77`.

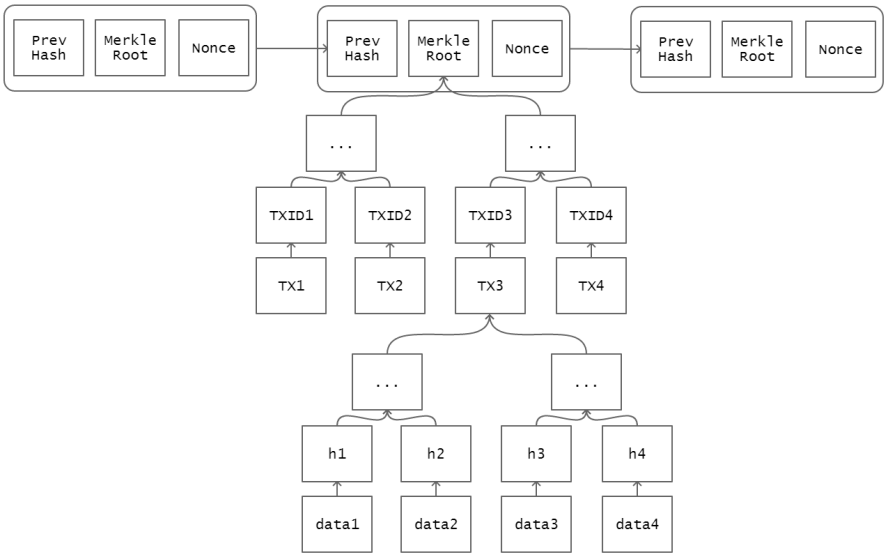

Figure 4.2: Scalability solution with *OpenTimestamps*.

Source: [10]

In addition to this, *OpenTimestamps* comes with a system of *aggregation servers*[10], publicly available "meeting points", where anyone can submit a digest to be timestamped, as Peter Todd said in [18]. As its name suggests, an aggregation server collects documents' hash values in a pending list of digests and periodically *aggregates* these digests into a single merkle tree. Notice that the aggregation server learns nothing about the original documents, it only collects "meaningless" digests[11]. In addition each file is protected by a nonce, thus files are not directly connected in any way. Then, the root of that tree is timestamped with Bitcoin. But there is a little price to pay, aggregation servers are obviously centralized and represent single points of failure because they could go offline, stopping the service. However this represents just a little inconvenience, aggregation 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.

[10]Examples of public *aggregation servers* are: `a.pool.opentimestamps.org` and `b.pool.opentimestamps.org`.

[11]In the sense that there is no way to retrieve the original document from its hash value.

The real downside is that aggregation servers are extremely efficient but not convenient. To obtain a proof, even if for more documents at once, you have to wait for the transaction committing to the document to confirm, that is 10 minutes on average if it is included in the next block. That is too much time in many cases, for example in a timestamped e-mails exchange.

But there is a compromise to address this problem which allows any user to receive such proof almost instantly: public *calendar servers*. A *calendar server* grants remote access to a *calendar*, a collection of timestamps. Calendar servers and aggregation servers work in conjunction: rather than timestamping the tip[12] of the merkle tree of digests directly with Bitcoin, aggregation servers aggregate pending digests in a one second interval into a merkle tree and submit the tip of that tree to a public calendar server. The latter makes the promise that every submitted merkletip 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, as aggregation servers before, a central point of failure. In fact, a proof made with the aid of a calendar server is called *incomplete*. However, once the Bitcoin blockchain has definitely completed the timestamp, such incomplete proof can be *upgraded*, adding the path up to the block header merkletip, but we will go into details later in this chapter. Algorithm 4.1, a slight modification of the one proposed by L. Comandini in his master thesis work [10], perfectly describes how calendar and aggregation servers cooperate.

Algorithm 4.1 Cooperation between aggregation and calendar servers

1: clients send data to timestamp to the aggregator[13] ▷ timestamp requests
2: aggregator composes a merkle tree of digests each second ▷ aggregation
3: aggregator sends to calendar the merkletip to timestamp
4: calendar promise he will timestamp the tip ▷ pending attestation
5: aggregator sends back to clients incomplete proof until the tip
6: calendar aggregates pending tips in a merkle tree ▷ aggregation
7: calendar sends a transaction including a merkletip ▷ timestamp
8: the transaction gets confirmed ▷ attestation complete
9: clients ask to the calendar to upgrade the timestamp ▷ upgrade
10: calendar sends back clients the complete proofs
11: clients verify their proofs ▷ verification

[12]We use the word *tip* as a synonym of *root*, hence whenever the word *merkletip* will appear, it indicates the root of that merkle tree.

One last question needs to be answered: who really pays for the transactions in which digests to be timestamped are pushed in? A calendar spends bitcoins from its own wallet, making *replace-by-fee*[14] transactions in order to spend a fixed amount each day. Briefly, the calendar makes a transaction with the lowest possible fees, recursively increasing that amount by a parameter a, fixed by who effectively runs the calendar server, until the transaction enters in a block. This result in a fixed amount of $144 \cdot a$ fees spent by the calendar each day, counting a block every 10 minutes on average. It could happen that some timestamps takes too long to be completed, a price that a user willingly pay for a service that allows completely free timestamping.

What remains to be done is to learn how to practically timestamp a document with *OpenTimestamps*, which provides users multiple and easy way to create and independently verify timestamps, directly from the official web page [4] or with command-line tools. We will now provide a step-by-step guide to use both the web page and the Python version of the client[15].

4.3 OpenTimestamps Python Client

OpenTimestamps client is a command-line[16] tool to create and validate timestamp proofs, with the Bitcoin blockchain as a notary. This section is aimed to be a step-by-step tutorial to fully manage the Python release of this client[17].

Before installing the *OpenTimestamps* client, we must specify that while you can *create* timestamps without a local Bitcoin Core node, to *verify* proofs you need one[18] (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.

[13] The term *aggregator* stands for aggregation server.

[14] In Bitcoin, unconfirmed transactions (yet in the mempool) can be modified and reissued with higher fees for the miners. This leads to a higher probability for that transaction to be mined.

[15] *OpenTimestamps* client comes in many programming languages: Python, Java, JavaScript and Rust. Here we present only the Python version because the main reference library of the project is written in Python.

[16] Here we are supposing a Unix-based operative system.

[17] The presented guide refers to the Git repository *opentimestamps-client* from the *OpenTimestamps* source code [3].

[18] For the details about running a Bitcoin node on your local machine, see Appendix A.

- 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
```

- Pick a file or create a new one to timestamp:

```
$ cat > filename.txt
bla bla bla ... (write what you want)
(type CTRL+D to close the file and return to
  command prompt)
```

- Timestamp your file:

```
$ ots stamp filename.txt
Submitting to remote calendar
  https://a.pool.opentimestamps.org
Submitting to remote calendar
  https://b.pool.opentimestamps.org
Submitting to remote calendar
  https://a.pool.eternitywall.com
Submitting to remote calendar
  https://ots.btc.catallaxy.com
```

  Nice! Your timestamp request is submitted to the main public calendar servers.

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

- Verify the proof (local Bitcoin Core node needed):

```
$ ots verify filename.txt.ots
Assuming target filename is 'filename.txt'
Calendar https://bob.btc.calendar.opentimestamps.
  org: Pending confirmation in Bitcoin blockchain
```

```
Calendar https://alice.btc.calendar.opentimestamps.
  org: Pending confirmation in Bitcoin blockchain
Calendar https://btc.calendar.catallaxy.com:
  Pending confirmation in Bitcoin blockchain
Calendar https://finney.calendar.eternitywall.com:
  Pending confirmation in Bitcoin blockchain
```

Notice that you can't verify immediately the aforementioned proof. In fact, the lines saying "pending confirmation" specify that 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). You can check for the current status of your proof file with the `info` command.

- Get detailed proof information:

```
$ ots info filename.txt.ots
File sha256 hash:
  13a1f687ddb7c1a0b12adeb708a2b464c9
  cfc24d5c9def4fa775cca81162feed
Timestamp:
append 5669aa818cc1c5bf8129d469f884fe79
sha256
 -> append 0f2a11ae083c66674d9d0d3da049079e
    sha256
    prepend
  f6836f41e86344eb7e4da03fe57c3e998d13594b2a27b4
  ccb86cef8ac19cf69e
    sha256
    prepend 5c502267
    append 2c767f767b02b45c
    verify PendingAttestation
  ('https://bob.btc.calendar.opentimestamps.org')
 -> append 675406ae56e9e40809ac587ee009067c
    sha256
    append
  7690b939b5a663ea311992aadf2f182424cea544bbfb2a
  837ae366ba5cc1ecf4
    sha256
    prepend 5c502266
    append 1269e2007cc47092
    verify PendingAttestation
  ('https://alice.btc.calendar.opentimestamps.org')
```

```
 -> append 9b10d7ec7f0af842407d2d7cbebac9ad
    sha256
    prepend 5c502267
    append da1d41ee8803dff5
    verify PendingAttestation
  ('https://btc.calendar.catallaxy.com')
 -> append f923446e73e3a503121a56006723a6c1
    sha256
    append c35e379ef7097a65fdbfb2a594a93c75
    sha256
    prepend 5c502266
    append c47a3b08bfc2315e
    verify PendingAttestation
  ('https://finney.calendar.eternitywall.com')
```

Incomplete timestamps can be upgraded using the `upgrade` command which adds the path to the relative block header merkleroot to the proof. Upgrading a proof isn't always available: there must be at least one completed attestation[19].

- Upgrade the proof:

```
$ ots upgrade filename.txt.ots
Got 1 attestation(s) from
  https://bob.btc.calendar.opentimestamps.org
Got 1 attestation(s) from
  https://alice.btc.calendar.opentimestamps.org
Got 1 attestation(s) from
  https://btc.calendar.catallaxy.com
Got 1 attestation(s) from
  https://finney.calendar.eternitywall.com
Success! Timestamp complete
```

In this case the timestamp is fully confirmed by the Bitcoin blockchain, so upgrading the proof succeeded.

Now that we know for sure that timestamping succeeded, you only left to use the `verify` command another time to check which block attested your timestamp (add the flag `-v` to be more verbose).

[19] That is, at least one of the four public calendars in question must have received an attestation from the Bitcoin blockchain.

```
$ ots -v verify filename.txt.ots
Assuming target filename is 'filename.txt'
Hashing file, algorithm sha256
Got digest
  13a1f687ddb7c1a0b12adeb708a2b464c9cfc24d5c9def
  4fa775cca81162feed
Attestation block hash:
  000000000000000013e603482
  1d3d85e8a8ff0217482e4b2ab602ec7ab6ce7
Success! Bitcoin block 560604 attests existence as
  of 2019-01-29 CET
```

4.4 OpenTimestamps Web Interface

Rather than the more complicated client, the easiest way to perform a timestamp with *OpenTimestamps* is using the web interface on the official website [4]. It obviously allows the same actions as the client does, just interacting with the "Stamp & Verify" box. Simply, drop a file in the box to *stamp* it and the service will automatically download its proof, named as the file is with a `.ots` extension. The hash of the document will be calculated inside your browser, thus not requiring the user to distribute his original document to any third party, preserving privacy.

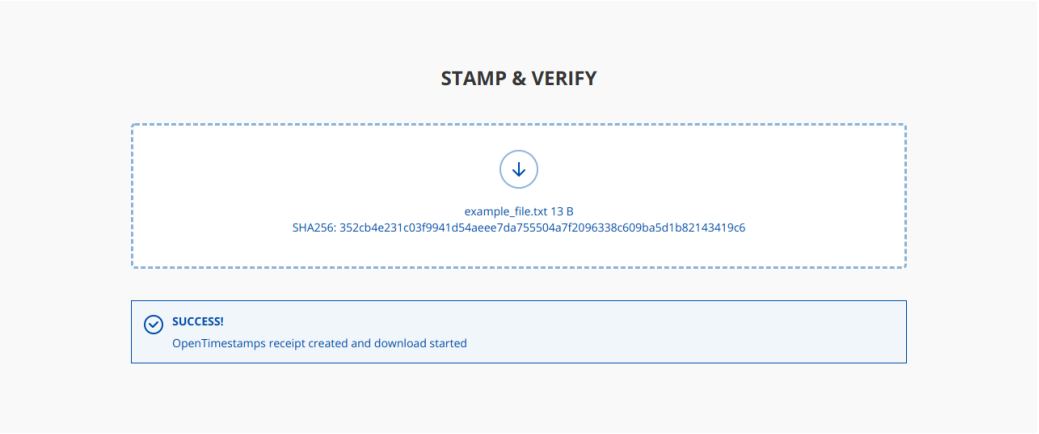

Figure 4.3: Stamping

Alternatively, you can drop a `.ots` file to *verify*. Here the underlying code will detect if the file is yet in the *incomplete* state and performs an *upgrade*, automatically downloading the upgraded proof file. Otherwise, if the timestamp is already completed, it verifies such proof. Notice that the original document is required to start the verification process.

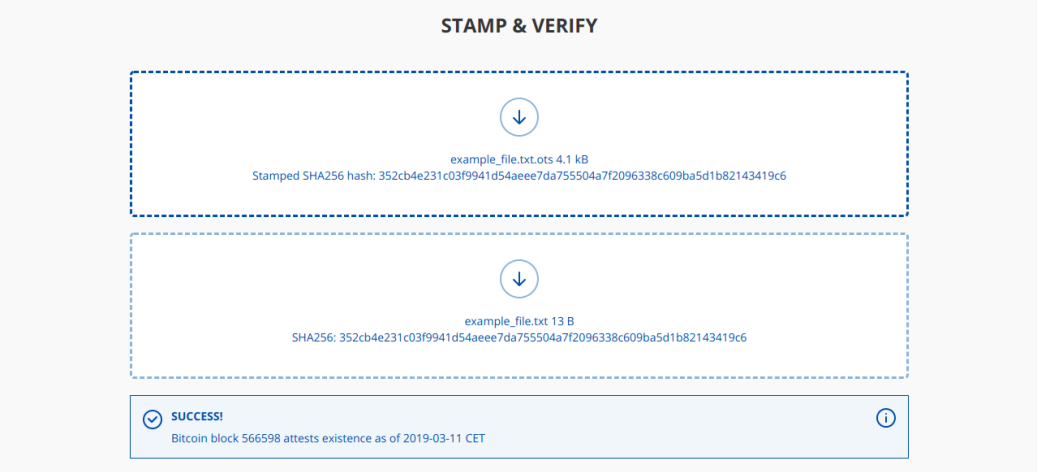

Figure 4.4: Verifying

However, the web interface comes with a downside. Browsers applications like this one have some security restrictions, inhibiting a direct access to the local Bitcoin configuration file[20], thus preventing a user to make RPC calls to his own local Bitcoin node in order to verify a timestamp proof. Hence a user must trust the public block explorers that *OpenTimestamps* uses to verify such proofs. Summing up, verification is not fully decentralized. Finally, clicking the *info* button in the low right corner will open a new web page that parse the proof in a nice representation (like the `info` command does for the client version, but with some added graphics).

[20]We refer to Appendix A for a detailed explanation about running a Bitcoin node, what RPC calls are and the main ones used in Bitcoin.

Chapter 5

Blockchain Notarization: A Practical Use Case

This Chapter is aimed to be a technical report of a project in which the author, in partnership with DGI (Digital Gold Institute) and ANIA (Associazione Nazionale fra le Imprese Assicuratrici), worked during the draft of this thesis.

5.1 Executive Summary

The purpose of this project work is to provide future clients a fully operating timestamping service, to enforce the notarization of digital documents which, as we already know, is traditionally achieved by certification authorities (CA), with a decentralized solution that is resilient even if the security of such CA is violated from the inside.

The timestamping service is implemented, with some extensions and modification on the basis of need, according to the *OpenTimestamps* protocol, because of its high scalability and low maintenance cost, all nice features that we already saw in the description of such protocol in Section 4.2. In addition, the proposed solution is independent of any provider, being *OpenTimestamps* an open-source project. It could be made available to customers for free or in form of a subscription service, with specific service level agreements, in order to provide additional features like the custody of any user's timestamp proof, usually charged to the latter. For example, any insurance company could make use of this solution to grant authenticity of its associates' insurance policies or, more generally, whenever a digital signature is involved. We

remind that *OpenTimestamps* also provides all the guarantees for an independent audit: a user can verify the validity of its timestamp proof without the need of the calendar server which actually maintains the service up, just by querying a local Bitcoin node or a public block explorer. Next we will move on the technical details regarding how this service is built, specifying when the *OpenTimestamps* protocol is improved.

5.2 Architecture of the Solution

The architecture of the solution consists of three servers located in cloud, and accessible from the outside via network elements (DNS server, reverse proxy, firewall, etc.) able to transparently remap host names under the aniasafe.it domain on local IPs to the DMZ (demilitarized zone)[1].

A user can easily connect to the public web server via browser at `https://timestamp.aniasafe.it` and upload a document to timestamp[2]. It is directly from browser (thanks to the underlying JavaScript code) that the timestamp request is sent to the calendar servers to be processed. The corresponding timestamp proof is automatically downloaded on the user's local device and can be verified on the same web page, simply uploading the `.ots` file. In that specific case, the browser will query public block explorers[3] in order to perform the verification procedure.

Before getting into technical details regarding the code base that underlies this solution, we present two explanatory figures. On one hand, Figure 5.1 is aimed to give an abstract idea of the whole timestamping process, on the other hand Figure 5.2 refers to the verification procedure.

[1] More precisely, a demilitarized zone is a physical or logical subnetwork that contains and exposes an organization's external-facing services to an untrusted network, usually a larger one like the Internet. All the details here: `https://en.wikipedia.org/wiki/DMZ_(computing)`.

[2] We remind that what is really timestamped is the *hash value* of such document, automatically computed on user's local device, without harming privacy. We refer to Chapter 4 for all the details.

[3] As specified in Section 4.4, timestamping services deployed as browser applications cannot query a local Bitcoin node and have to trust public block explorers.

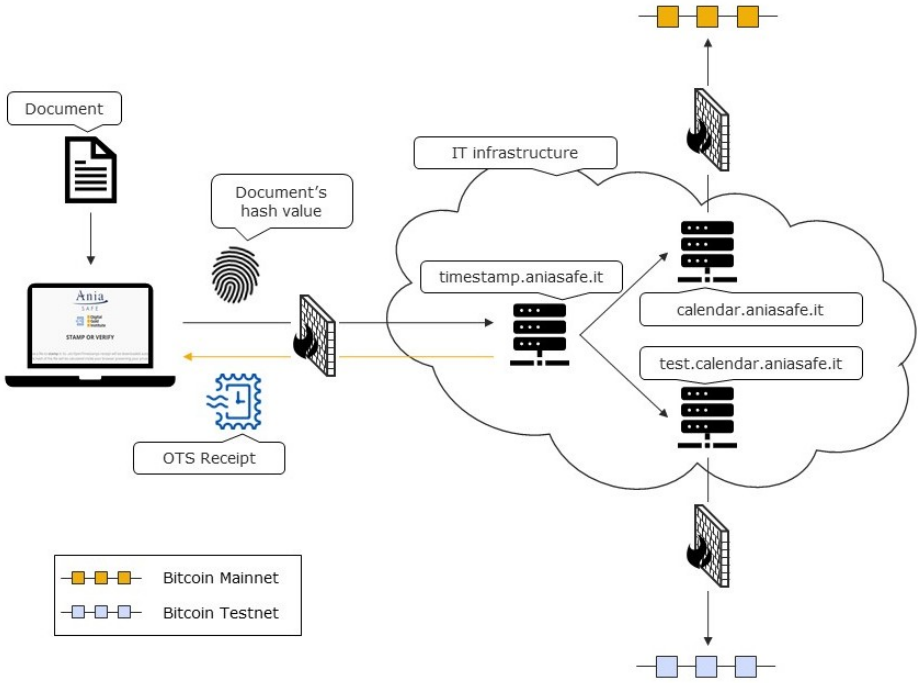

Figure 5.1: Architecture of the timestamping process

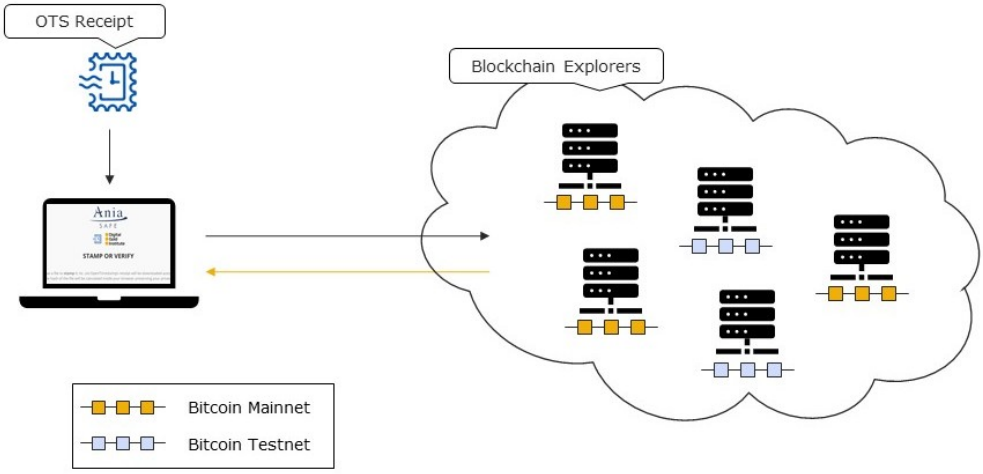

Figure 5.2: Architecture of the verification process

5.3 Technical Details

The three servers introduced before are divided in a front-end server (`https://timestamp.aniasafe.it`) which hosts the public web page, and two back-end servers, the first one running a calendar server connected to a local[4] Bitcoin node in mainnet mode (`https://calendar.aniasafe.it`), the latter connected to a local one in testnet mode (`https://test-calendar.aniasafe.it`).

As we already know, the whole timestamping service is implemented according to the open-source *OpenTimestamps* protocol. For a better understanding we will separate the client side from the server one, clarifying their functioning.

5.3.1 Client Side

The core of the client side is the underlying JavaScript library [2], which defines the main functions invoked when a user interacts with the web interface hosted by the front-end server: *stamp*, *verifyTimestamp* and *upgrade*.

- *stamp*: it takes two parameters in input: *detaches* that is the array of detached files to stamp and *options*, which specifies the calendars to send a timestamp request. First, the function makes the parsing of input parameters to check if they are in the right format. Then it invokes *makeMerkleTree* that, as the name suggests, builds a merkle tree of such files and passes the root of that tree to *createTimestamp*, a function that composes a timestamp and submits it to the list of calendars specified on the options parameter.

- *verifyTimestamp*: it takes two parameters in input too: a `.ots` file (the proof to be verified) and *options* which, for any queried chain (f.e. Bitcoin mainnet, Bitcoin testnet, Litecoin etc.), specifies a list of block explorers and their types in order to fully manage the corresponding API[5] in the right way. The function parses the file in input and, for any attestation found, it verifies the status of such attestation calling *verifyAttestation*. If the attestation is in *Pending* or *Unknown* status, a

[4] In the sense that the calendar server and the Bitcoin node run on the same machine.

[5] An Application Programming Interface (API) is a set of subroutine definitions, communication protocols, and tools for building software. In general terms, it is a set of clearly defined methods of communication among various components. More details here: `https://en.wikipedia.org/wiki/Application_programming_interface`.

warning message will notify that such attestation is incomplete or not recognized, in the sense that the tag relative to the chain used to stamp is not known. Otherwise, if the attestation is considered valid, first the function tries to reach a local node if present[6], else the verification proceeds with the help of one or more block explorers. After a check of the chain in which the file is attested, it will be created an object named *liteOptions* which extracts from the input parameter *options* only the piece relative to the chain in question. Finally this new object is passed to the function *liteVerify*[7], which generates an instance of the class *ExplorerList*, that is a list of block explorers such function will interact to, in accordance to the ones specified in *liteOptions*. These block explorers will query the blockchain with two calls, the first asking for the relative block hash, the latter, using that hash value as input, retrieves all the information of the block needed for the verification process.

5.3.2 Server Side

5.3.3 Extensions to the protocol

The front-end server hosts the web interface. Here, the usage of the timestamp service is almost identical to the one deployed in the official web page. You can drag & drop a document to timestamp or a `.ots` file to upgrade if the timestamp is incomplete, or to verify if complete. The only modification actually visible to the eye is that the original file is no more required to start the verification process. You can check if the timestamped hash matches the original in a separate box. This allows to delegate the verification process to any third party without harming privacy, just distributing the `.ots` file, which contains no information about the content of the original document. If necessary, the owner of the original document will prove the veracity of such document by showing its hash value.

A simplified example of the options format could be:

```
const options = {
  bitcoin: {
    explorers: [
```

[6]We remind that verifying with a local node is always the best practice, having not to trust any third party.

[7]Lite or "light" because it verifies one attestaion at a time.

```
        {url: 'https://blockstream.info/api', type: '
blockstream'},
        {url: 'https://blockexplorer.com/api', type:
'insight'}
      ]
    },
    litecoin: {
      explorers: [
        {url: 'https://ltc-bitcore1.trezor.io/api',
type: 'insight'},
        {url: 'https://insight.litecore.io/api', type
: 'insight'}
      ]
    }
  }
```

Chapter 6

Conclusions

Appendix A

Bitcoin from the command line

This appendix is aimed at providing a technical walkthrough the Bitcoin network from the command line point of view, supposing a Unix-like operating system. Initially, we will configure a machine to launch a Bitcoin node; then we will introduce some of the most important RPC calls[1] in order to perform basic operations.

A.1 Running a Bitcoin Core node

To get started with Bitcoin, you first need to join the network running a node. Following these simple steps[2] will make the first experience a lot easier.

- Setup variables (for an easy installation):

```
$ export BITCOIN=bitcoin-core-0.17.1
```

- Download relevant files (Remark: every time you find username replace it with your personal one):

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

[1] You can find the original Bitcoin client/API calls list here: `https://en.bitcoin.it/wiki/Original_Bitcoin_client/API_calls_list`.

[2] The presented guide is a simplified version of a very detailed open-source guide by C. Allen [5], conceived for a local installation of Bitcoin Core.

```
$ wget https://bitcoin.org/laanwj-releases.asc -O ~
  username/laanwj-releases.asc
```

- Verify Bitcoin signature (in order to verify that your Bitcoin setup is authentic):

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

  Amongst the info you get back from the last command should be a line telling "Good signature", do not care the rest.

- Verify Bitcoin SHA (Secure Hash Algorithm) of the .tar file against the expected one:

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

  From this verification you must obtain the same number.

- Install Bitcoin:

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

- Create the configuration file:

  First, create the directory you want to put your data into:

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

  Then create the configuration file (as you need it):

```
$ cat >> ~username/.bitcoin/bitcoin.conf << EOF
# Accept command line and JSON-RPC commands
server=1
# Username for JSON-RPC connections
rpcuser=invent_a_username_here
# Password for JSON-RPC connections
rpcpassword=invent_a_long_pwd_here
# Enable Testnet chain mode
testnet=1
EOF
```

The *bitcoin.conf* file is the default file which Bitcoin daemon reads when launched, and it contains instructions to configure the node (e.g. mainnet, testnet or regtest mode). In this example we define a *rpcuser* and *rpcpassword* to enable RPC, and we tell the node to synchronize with the Bitcoin Testnet chain.

Finally, limit the permissions to your configuration file (not really needed, but more secure):

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

- Start the daemon[3]:

```
$ bitcoind -daemon
```

We are now ready to be part of the Bitcoin network !
In the next section we will learn how to manage the Bitcoin command line interface.

A.2 Bitcoin-cli: command line interface

In this section we will answer the following question: "*What is RPC and which calls are the most used in Bitcoin?*".

[3]In multitasking computer operating systems, a *daemon* is a computer program that runs as a background process, rather than being under direct control of an interactive user. Traditionally, the process names of a daemon end with the letter *d*; `bitcoind` is the one which implements the Bitcoin protocol for remote procedure call (RPC) use.

Without further going into details regarding computer science, a RPC[4] (Remote Procedure Call) is a powerful technique for constructing distributed, client-server based applications. It is based on the extension of the conventional procedure calling, so that the called procedure need not exist in the same address space as the calling procedure. The two processes may be on the same system, or they may be on different systems with a network connecting them. Our interest is for the `bitcoin-cli`, a handy interface that lets the user send commands to the `bitcoind`. More specifically, it lets you send RPC commands to the `bitcoind`. Generally, the `bitcoin-cli` interface is much more clean and user friendly than trying to send RPC commands by hand, using `curl` or some other method.

The first thing to do for a beginner user is to look for the full list of calls invoking the `help` command:

```
$ bitcoin-cli help

== Blockchain ==
getbestblockhash
getblock "blockhash" ( verbosity )
getblockchaininfo
getblockcount
getblockhash height
getblockheader "hash" ( verbose )
getblockstats hash_or_height ( stats )
getchaintips
getchaintxstats ( nblocks blockhash )
getdifficulty
getmempoolancestors txid (verbose)
getmempooldescendants txid (verbose)
getmempoolentry txid
getmempoolinfo
getrawmempool ( verbose )
gettxout "txid" n ( include_mempool )
gettxoutproof ["txid",...] ( blockhash )
gettxoutsetinfo
preciousblock "blockhash"
pruneblockchain
savemempool
scantxoutset <action> ( <scanobjects> )
verifychain ( checklevel nblocks )
```

[4]For more details, see `https://en.wikipedia.org/wiki/Remote_procedure_call`.

```
verifytxoutproof "proof"

== Control ==
getmemoryinfo ("mode")
help ( "command" )
logging ( <include> <exclude> )
stop
uptime

== Generating ==
generate nblocks ( maxtries )
generatetoaddress nblocks address (maxtries)

== Mining ==
getblocktemplate ( TemplateRequest )
getmininginfo
getnetworkhashps ( nblocks height )
prioritisetransaction <txid> <dummy value> <fee delta>
submitblock "hexdata"  ( "dummy" )

== Network ==
addnode "node" "add|remove|onetry"
clearbanned
disconnectnode "[address]" [nodeid]
getaddednodeinfo ( "node" )
getconnectioncount
getnettotals
getnetworkinfo
getpeerinfo
listbanned
ping
setban "subnet" "add|remove" (bantime) (absolute)
setnetworkactive true|false

== Rawtransactions ==
combinepsbt ["psbt",...]
combinerawtransaction ["hexstring",...]
converttopsbt "hexstring" ( permitsigdata iswitness )
createpsbt [{"txid":"id","vout":n},...] [{"address":
  amount},{"data":"hex"},...] ( locktime ) (
  replaceable )
createrawtransaction [{"txid":"id","vout":n},...] [{"
  address":amount},{"data":"hex"},...] ( locktime ) (
```

```
  replaceable )
decodepsbt "psbt"
decoderawtransaction "hexstring" ( iswitness )
decodescript "hexstring"
finalizepsbt "psbt" ( extract )
fundrawtransaction "hexstring" ( options iswitness )
getrawtransaction "txid" ( verbose "blockhash" )
sendrawtransaction "hexstring" ( allowhighfees )
signrawtransaction "hexstring" ( [{"txid":"id","vout":n
  ,"scriptPubKey":"hex","redeemScript":"hex"},...] ["
  privatekey1",...] sighashtype )
signrawtransactionwithkey "hexstring" ["privatekey1
  ",...] ( [{"txid":"id","vout":n,"scriptPubKey":"hex
  ","redeemScript":"hex"},...] sighashtype )
testmempoolaccept ["rawtxs"] ( allowhighfees )

== Util ==
createmultisig nrequired ["key",...] ( "address_type" )
estimatesmartfee conf_target ("estimate_mode")
signmessagewithprivkey "privkey" "message"
validateaddress "address"
verifymessage "address" "signature" "message"

== Wallet ==
abandontransaction "txid"
abortrescan
addmultisigaddress nrequired ["key",...] ( "label" "
  address_type" )
backupwallet "destination"
bumpfee "txid" ( options )
createwallet "wallet_name" ( disable_private_keys )
dumpprivkey "address"
dumpwallet "filename"
encryptwallet "passphrase"
getaccount (Deprecated, will be removed in V0.18. To
  use this command, start bitcoind with -deprecatedrpc=
  accounts)
getaccountaddress (Deprecated, will be removed in V0
  .18. To use this command, start bitcoind with -
  deprecatedrpc=accounts)
getaddressbyaccount (Deprecated, will be removed in V0
  .18. To use this command, start bitcoind with -
  deprecatedrpc=accounts)
```

```
getaddressesbylabel "label"
getaddressinfo "address"
getbalance ( "(dummy)" minconf include_watchonly )
getnewaddress ( "label" "address_type" )
getrawchangeaddress ( "address_type" )
getreceivedbyaccount (Deprecated, will be removed in V0
  .18. To use this command, start bitcoind with -
  deprecatedrpc=accounts)
getreceivedbyaddress "address" ( minconf )
gettransaction "txid" ( include_watchonly )
getunconfirmedbalance
getwalletinfo
importaddress "address" ( "label" rescan p2sh )
importmulti "requests" ( "options" )
importprivkey "privkey" ( "label" ) ( rescan )
importprunedfunds
importpubkey "pubkey" ( "label" rescan )
importwallet "filename"
keypoolrefill ( newsize )
listaccounts (Deprecated, will be removed in V0.18. To
  use this command, start bitcoind with -deprecatedrpc=
  accounts)
listaddressgroupings
listlabels ( "purpose" )
listlockunspent
listreceivedbyaccount (Deprecated, will be removed in
  V0.18. To use this command, start bitcoind with -
  deprecatedrpc=accounts)
listreceivedbyaddress ( minconf include_empty
  include_watchonly address_filter )
listsinceblock ( "blockhash" target_confirmations
  include_watchonly include_removed )
listtransactions (dummy count skip include_watchonly)
listunspent ( minconf maxconf  ["addresses",...] [
  include_unsafe] [query_options])
listwallets
loadwallet "filename"
lockunspent unlock ([{"txid":"txid","vout":n},...])
move (Deprecated, will be removed in V0.18. To use this
   command, start bitcoind with -deprecatedrpc=accounts
  )
removeprunedfunds "txid"
rescanblockchain ("start_height") ("stop_height")
```

```
sendfrom (Deprecated, will be removed in V0.18. To use
  this command, start bitcoind with -deprecatedrpc=
  accounts)
sendmany "" {"address":amount,...} ( minconf "comment"
  ["address",...] replaceable conf_target "
  estimate_mode")
sendtoaddress "address" amount ( "comment" "comment_to"
   subtractfeefromamount replaceable conf_target "
  estimate_mode")
setaccount (Deprecated, will be removed in V0.18. To
  use this command, start bitcoind with -deprecatedrpc=
  accounts)
sethdseed ( "newkeypool" "seed" )
settxfee amount
signmessage "address" "message"
signrawtransactionwithwallet "hexstring" ( [{"txid":"id
  ","vout":n,"scriptPubKey":"hex","redeemScript":"hex
  "},...] sighashtype )
unloadwallet ( "wallet_name" )
walletcreatefundedpsbt [{"txid":"id","vout":n},...] [{"
  address":amount},{"data":"hex"},...] ( locktime ) (
  replaceable ) ( options bip32derivs )
walletlock
walletpassphrase "passphrase" timeout
walletpassphrasechange "oldpassphrase" "newpassphrase"
walletprocesspsbt "psbt" ( sign "sighashtype"
  bip32derivs )

== Zmq ==
getzmqnotifications
```

You can also type `bitcoin-cli help [command]` to be even more extensive about that command. For example:

```
$ bitcoin-cli help getmininginfo

getmininginfo

Returns a json object containing mining-related
  information.
Result:
{
```

```
  "blocks": nnn,             (numeric) The current
  block
  "currentblockweight": nnn, (numeric) The last block
  weight
  "currentblocktx": nnn,     (numeric) The last block
  transaction
  "difficulty": xxx.xxxxx    (numeric) The current
  difficulty
  "networkhashps": nnn,      (numeric) The network
  hashes per second
  "pooledtx": n              (numeric) The size of the
  mempool
  "chain": "xxxx",           (string) current network
  name as defined in BIP70 (main, test, regtest)
  "warnings": "..."          (string) any network and
  blockchain warnings
}

Examples:
> bitcoin-cli getmininginfo
> curl --user myusername --data-binary '{"jsonrpc":
  "1.0", "id":"curltest", "method": "getmininginfo", "
  params": [] }' -H 'content-type: text/plain;' http
  ://127.0.0.1:8332/
```

Here is a list of the most general commands to get additional information about bitcoin data:

```
$ bitcoin-cli getblockchaininfo
$ bitcoin-cli getmininginfo
$ bitcoin-cli getnetworkinfo
$ bitcoin-cli getnettotals
$ bitcoin-cli getwalletinfo
```

For example `bitcoin-cli getwalletinfo` gives the user precise information about the bitcoin wallet, like the confirmed and unconfirmed balance (nota a pie' pagina per spiegare cosa sia unconfirmed balance), and the total number of transactions in the wallet.

```
$ bitcoin-cli getwalletinfo
{
 "walletname": "",
```

```
  "walletversion": 169900,
  "balance": 0.07825304,
  "unconfirmed_balance": 0.00000000,
  "immature_balance": 0.00000000,
  "txcount": 1,
  "keypoololdest": 1544714806,
  "keypoolsize": 999,
  "keypoolsize_hd_internal": 1000,
  "paytxfee": 0.00000000,
  "hdseedid": "972e6ac8d0104fec0d0e6c052d3876ada965c745
   ",
  "hdmasterkeyid": "972
   e6ac8d0104fec0d0e6c052d3876ada965c745",
  "private_keys_enabled": true
}
```

Bibliography

[1] Bitcoin developer guide. https://bitcoin.org/en/developer-reference.

[2] Javascript library source code. https://github.com/federicoon/javascript-opentimestamps.

[3] Opentimestamps source code. https://github.com/opentimestamps.

[4] Opentimestamps website. https://opentimestamps.org/.

[5] Allen, C. Learning-bitcoin-from-the-command-line source code. https://github.com/ChristopherA/Learning-Bitcoin-from-the-Command-Line.

[6] Antonopoulos, A. M. *Mastering Bitcoin: Programming the Open Blockchain*, 2nd ed. O'Reilly Media, Inc., 2017.

[7] Back, A. Hashcash - a denial of service counter-measure. Tech. rep., 2002.

[8] Bayer, D., Haber, S., and Stornetta, W. S. Improving the efficiency and reliability of digital time-stamping. In *Sequences II: Methods in Communication, Security and Computer Science* (1993), Springer-Verlag, pp. 329–334.

[9] Ben-Or, M. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1983), PODC '83, ACM, pp. 27–30.

[10] Comandini, L. sign-to-contract: how to achieve trustless digital timestamping with zero marginal cost. Master's thesis, Politecnico di Milano, 2018.

[11] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (Apr. 1985), 374–382.

[12] Haber, S., and Stornetta, W. S. How to time-stamp a digital document. *Journal of Cryptology 3* (1991), 99–111.

[13] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 382–401.

[14] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, 2009.

[15] Narayanan, A., Bonneau, J., Felten, E., Miller, A., and Goldfeder, S. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, Princeton, NJ, USA, 2016.

[16] Paar, C., and Pelzl, J. *Understanding Cryptography: A Textbook for Students and Practitioners*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[17] Todd, P. Interpreting ntime for the purpose of bitcoin-attested timestamps. `https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2016-September/013120.html`, 2016.

[18] Todd, P. Opentimestamps: Scalable, trust-minimized, distributed timestamping with bitcoin. `https://petertodd.org/2016/opentimestamps-announcement`, 2016.

[19] Todd, P. Sha1 is broken, but it's still good enough for opentimestamps. `https://petertodd.org/2017/sha1-and-opentimestamps-proofs`, 2017.

[20] Wattenhofer, R. *The Science of the Blockchain*, 1st ed. CreateSpace Independent Publishing Platform, USA, 2016.