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

The secret to happiness is freedom
and the secret to freedom is courage.

Thucydides

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

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 multicore processors or computing clusters.

On the other side, every day each of us operates with a distributed system; just think about a modern smartphone: it can share multiple data on the cloud and can contain multiple processing devices. Otherwise withdrawing money at the ATM is a perfect example of usage of a distributed system.

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 much-wanted boost to performance.
- Fault Tolerance: A cluster of multiple machines is inherently more fault-tolerant than a single one.
- Reliability: Data is replicated on different machines in order to prevent loss.
- Availability: Data is replicated on different machines to minimize latency and grant fast access.

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

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

2.2 Consensus

(reworka tutto)
As we learned before, 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, and atomic broadcasts. The real world applications include clock synchronization, PageRank, opinion formation, smart power grids, state estimation, control of UAVs (and multiple robots/agents in general), load balancing and others. Let us get in formal definitions:

Definition 2.2.1. (node). *We call a single actor in the system 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.*

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

Definition 2.2.2. (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 an input value v_i. The nodes must decide for one of those values, satisfying the following properties:*

- *Agreement: All correct noes decide for the same value.*
- *Termination: All correct nodes terminate in finite time.*
- *Validity: The decision value must be the input value of a node.*

2.2.1 Two friends

Nota: preso da wattenhoefer.
Alice wants to arrange dinner with Bob, and since both of them are very reluctant to use the "call" functionality of their phones, she sends a text message suggesting to meet for dinner at 6pm. However, texting is unreliable, and Alice cannot be sure that the message arrives at Bob's phone, hence she will only go to the meeting point if she receives a confirmation message from Bob. But Bob cannot be sure that his confirmation message is received; if the confirmation is lost, Alice cannot determine if Bob did not even receive her suggestion, or if Bob's confirmation was lost. Therefore, Bob demands a confirmation message from Alice, to be sure that she will be there. But as this message can also be lost...

You can see that such a message exchange continues forever, if both Alice and Bob want to be sure that the other person will come to the meeting point!

2.2.2 Byzantine Agreement

It could happen that nodes in a distributed system do not just crash, instead they could behave arbitrarily or be voluntarily malevolent.

Definition 2.2.3. (byzantine node)[3]. *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.*

[3]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.

Thus, achieving consensus (as in Definition 2.2.2) in a system with byzantine nodes is much more harder. 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 2.2.4. (any-input validity)[4]. *The decision value must be the input 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 2.2.5. (correct-input validity). *The decision value must be the input 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:

Definition 2.2.6. (all-same validity). *If all correct nodes start with the same input 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 2.2.7. (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.*

[4]This is the validity definition we implicitly used for consensus, in Definition 2.2.2

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 asynchronous networks, in presence of byzantine nodes.

Definition 2.2.8. (asynchronous model). *In the asynchronus 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 2.2.9. (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.*

A famous algorithm by Ben-Or (more details in [6]) tries to solve asynchronous byzantine agreement.

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

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.

Hence, an impossibility result by Fisher, Lynch and Paterson[5] comes in help:

Theorem 2.2.2. *There is no deterministic algorithm which always achieves consensus in the asynchronous model, with* $f > 0$.

Due to this proven impossibility result, how Satoshi Nakamoto reaches consensus in Bitcoin, a decentralized, distributed, peer-to-peer network? Before delving into this clever resolution of the problem we need to strengthen our cryptographic background.

[5]The proof of Fisher, Lynch and Paterson [7], known as FLP and established in 1985, was awarded the 2001 PODC Influential Paper Award (now called Dijkstra Price)

Chapter 3

Cryptographic Background

Chapter 4

OpenTimestamps Protocol

Chapter 5

OpenTimestamps User Guide

Chapter 6

Proposed Extensions to the Protocol

Chapter 7

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 (taken from... metti la fonte) in order to perform basic operations. Lastly the OpenTimestamps protocol comes into play: we will practically see how to timestamp a document using the Python client and how to set up a calendar server.

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 will make the first experience a lot easier.

- Setup variables (for an easy installation):

```
$ export BITCOIN=bitcoin-core-0.17.0
```

- 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
$ 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. (metti una nota qui per il sito dei conf file).

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

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

- Start the daemon[1]:

```
$ 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?*".

Without further going into details regarding computer science[2], a RPC (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

[1] 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.

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

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 )
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": "972e6ac8d0104fec0d0e6c052d3876ada965c745",
```

```
  "private_keys_enabled": true
}
```

In the next sections we will deal with notarization, in particular we will get into technical details of *OpenTimestamps* protocol.

A.3 OpenTimestamps client

OpenTimestamps client is a command-line tool to perform stamping of documents and to verify *OpenTimestamps* proofs, using the Bitcoin blockchain as a timestamp notary.
After this step-by-step tutorial we will be able to fully manage the Python release of the *OpenTimestamps* client.

Before installing the *OpenTimestamps* client, we must remind that while you can *create* timestamps without a local Bitcoin Core node, to *verify* proofs you need one (a pruned node is fine too, metti la nota per spiegare perche').

Let us get into a step-by-step guide to fully manage *OpenTimestamps* client in all its features:

- 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)
CTRL+D (to close file and return to command prompt)
```

- Timestamp your example 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.

Now in your current directory you must have a file named `filename.txt.ots` that is the proof of your timestamp[3].

- 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 above-mentioned proof: it takes a few hours for the timestamp to get confirmed by the Bitcoin blockchain; *OpenTimestamps* is not doing one transaction per timestamp.

Now the proof is *incomplete* because it needs the assistance of a remote calendar to verify; the calendar provides the path to the Bitcoin block header. You can check for the current status of your proof file with the `info` command.

- Get detailed proof information:

[3]More specifically, an *OpenTimestamps* proof is the path up to merkleroot of the block which contains the relative transaction.

```
$ 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 Bitcoin blockchain to the timestamp itself. Upgrading a proof isn't always available: there must be at least one completed attestation.

- 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 Bitcoin Blockchain, so upgrading the proof succeeded.

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

```
$ 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
```

Bibliography

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

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

[3] Opentimestamps website. `https://opentimestamps.org/`.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system, http://bitcoin.org/bitcoin.pdf, 2008.

[10] 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.

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

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

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

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

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