Stampery Blockchain Timestamping Architecture (BTA)

ADÁN SÁNCHEZ DE PEDRO CRESPO

STAMPERY, CTO

adan@stampery.com

LUIS IVAN CUENDE GARCÍA *

ARAGON, CEO

luis@aragon.one

Abstract

A method for timestamping, anchoring and certification of a virtually unlimited amount of data in one or more blockchains, focusing on scalability and cost-effectiveness while ensuring existence, integrity and ownership by using cryptographic proofs that are independently verifiable by anyone in the world without disclosure of the original data and without the intervention of the certifying party.

I. INTRODUCTION

Blockchains—understood as immutable, decentralized, distributed and unedited digital ledgers—have already proved their undeniable capability to serve as a universal support for a distributed monetary system.

Since the appearance of Bitcoin [1] and its revolutionary approach to digital cash, there have been many attempts to develop methods allowing anyone to leverage the immutability of the blockchain technology for purposes other than the transfer of currency.

The Bitcoin protocol itself provides an opcode [2] called OP_RETURN [3] that lets the participants of the network to embed arbitrary data into the transactions they broadcast to the Bitcoin blockchain since version 0.9.0 of *Bitcoin Core*, released in March 2014.

Since OP_RETURN was made available, tools like Proof of Existence [4], Stampery [5] or Tierion [6] have been making use of the Bitcoin blockchain to timestamp and verify data.

Nevertheless, the OP_RETURN method has several limitations that are intrinsic to the very specificities of the bitcoin protocol:

- In order to keep the average transaction and block size inside acceptable bounds, the maximum length for a single OP_RETURN operation is 80 bytes.

---

*Luis Iván—Stampery's former CTO and now one of our advisors—is credited here for being the original deviser and developer of BTA's very first version back in 2014.

- It is considered non-standard for a transaction to carry more than one OP_RETURN output.
- Due to the existing block size limit, the bitcoin network can handle up to 7 transactions per second under ideal circumstances, but the actual value is even lower.
- The fee that must be paid to the network in order to rest assured that all transactions are accepted and validated in a reasonable time span is high and may keep rising over time as Bitcoin value fluctuates. Broadcasting a transaction containing a single OP_RETURN with a certain guarantee that it will be written into the next block and confirmed nearly immediately can cost up to 120,000 satoshi (0.0012 Bitcoins), which at the time of the last review of this document roughly amounts to $3.12.

In a very first approach to overcoming some of those limitations, the aforementioned timestamping services do not actually embed the whole piece of data to be timestamped. Instead, they calculate a cryptographic hash of the data, which serves as a univocal identifier that has a length between 32 and 64 bytes and therefore fits into a single bitcoin transaction. This is what we refer as *data anchoring*, *blockchain timestamping* or simply *stamping*.

Nevertheless, there are some use cases which require bulk timestamping of hundreds or thousands of files; or timestamping the same piece of data over and over in its successive versions. This would require expending a huge amount of money in transaction fees and would quickly lead to saturation of the previously mentioned bitcoin block size limit. These consequences would obviously render the OP_RETURN method unsuitable for such use cases.

It is worth mentioning that other Proof-of-Work blockchains like Ethereum's or Litecoin's do not offer a definitive solution for those problems. Although they may benefit from lower transaction fees at the moment, given a huge volume of stamping actions, the limitations would be roughly the sames.

This technical paper proposes a method and architecture that allows (1) carrying out scalable and cost-effective blockchain data anchoring (overcoming all the already referred limitations), (2) generating irrefutable proofs of anchoring and (3) verification of the validity of such proofs by any individual or system in the world.

II.   Hashes aggregation

Many single hashes corresponding to different pieces of data or files can be compiled into a single hash by building a *binary tree* in which every leaf node is populated with each of the hashes while every non-leaf node is populated with the hash of the merger of its child nodes. This type of tree is commonly known as *Merkle tree* [7].
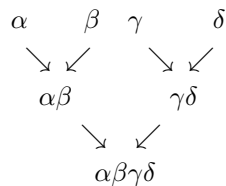

Figure 1: *Example of Merkle tree with 4 hashes as leaf nodes.*

Merkle trees have been used for decades to perform efficient and secure verification of the contents of large data structures [8]. Indeed, popular blockchain systems like Bitcoin, Ethereum or Litecoin rely on Merkle trees for gathering together transactions into every block of the chain.

The tip node in the tree—the hash resulting from aggregation of all the hashes in the leafs—is called *Merkle root* or simply *root*.

The binary hash chain [9] proving that a certain hash belongs to a tree—further discussed in Section III—is called *Merkle proof* or simply *proof*.
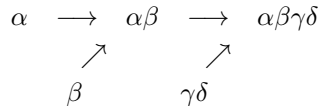
$$\alpha \;\longrightarrow\; \alpha\beta \;\longrightarrow\; \alpha\beta\gamma\delta$$
$$\beta \qquad\qquad \gamma\delta$$

Figure 2: *Binary hash chain representing the Merkle proof for the α hash in the tree depicted in* **Figure 1**. $\Pi(\alpha) = \{\beta, \gamma\delta\}$

By aggregating multiple hashes into a Merkle tree and then publishing only the root, it is possible to anchor large volumes of data in a blockchain with a single transaction, which dramatically reduces costs and avoids the aforementioned bottlenecks.

The algorithm followed to calculate the hash in each non-leaf node by joining its two child nodes is called a *mixing fuction*, *Merkle mixer* or simply *mixer*.

A very common mixer—used by the Stampery API and Chainpoint—can be expressed in the form $SHA256(\lambda + \rho)$ and represents the result of calculating the SHA-256 hash of the concatenation of the left child λ and the right child ρ. However, given that concatenation is a non-commutative binary operation ($\lambda + \rho \neq \rho + \lambda$), it makes necessary to distribute some additional bits of information along the Merkle proofs, being them (a) the position (index) in which each leaf hash was added, or (b) the order in which every pair of child nodes were concatenated.

To avoid the need to deal with those *order bits*, previous versions of BTA used a different approach to mixing functions: *commutative concatenation*.

Commutative concatenation, $\dot{+}$, of two values λ and P can be easily achieved by ordering them from lowest to highest before performing the concatenation. This way, $\lambda \dot{+} \rho = \rho \dot{+} \lambda$.

Former versions of BTA proposed the use of the *SHA-3 hash algorithm* as defined by FIPS-202[10] with a 512 bit key to ensure resistance to *length extension attacks* and many other theoretical vulnerabilities affecting SHA-2 and all other algorithms based on the *Merkle–Damgård construction* [11]. Nevertheless, the current versions of BTA and the Stampery API use SHA-256 by default for the sake of ease of integration into a wider range of platforms and devices.

Due to the way that binary trees work, the root is always connected to every leaf hash by a unique path with the exception of the cases when the number of nodes in one of the levels in the tree is odd. In such cases, in order to find a root that is connected to all leaf hashes, "orphan" nodes must be (a) promoted to the next level without applying any mixing function to its value; or (b) mixed with a random string of the same length of the actual hashes. This process is called *closing* a Merkle tree.

```
algorithm mixer is
  input: left hash λ,
         right hash ρ
  output: parent node hash π

  π ← SHA256(λ + ρ)
  return π
```

Figure 3: *The non-commutable mixing function used by the Stampery API, described in pseudocode.*
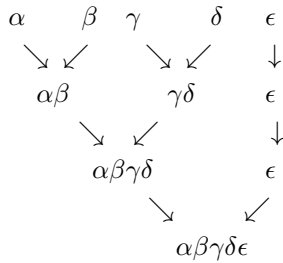

Figure 4: *Example of closing a Merkle tree with an odd number of leaf nodes: the ϵ node is promoted to its immediately upper level until it can be merged into the root.*

III.   ANCHORAGE PROCEDURES

As explained in the introduction, Bitcoin anchoring is not perfectly suitable for those cases in which a) latency is paramount; or b) a really fast or near-immediate confirmation is needed.

In the other hand, newer blockchain projects like Ethereum have still a lower hashing power—which translates to lower certainty and security—but they are much faster when it comes to the time needed to put a hash in a block.

Table 1: *Average block time comparison between highest capitalization blockchains.*

Chain	Average block time
Bitcoin	~10 minutes
Ethereum	~17 seconds
Ripple	~4 seconds
Litecoin	~2.5 minutes

BTA is designed to allow anchoring data into more than one blockchain, leveraging the advantages of each one. A recommended strategy is combining Bitcoin and Ethereum anchoring together to get certainty and responsiveness at the same time.

Anchoring to multiple chains and different ways to do so will be further discussed in Section VII.

i.   Bitcoin anchoring

Bitcoin anchoring is achieved by using the aforementioned OP_RETURN opcode.

The OP_RETURN data length limit used to be set to 40 bytes before February 2015. Then the limit was raised to 80 bytes [12], which allowed the use of longer hashing digests.

The hashes resulting from the recommended hashing algorithm in this document—SHA-2 in its 256 bit version—are 32 bytes long, so they fit perfectly into a single OP_RETURN instruction.

Addition of a OP_RETURN operation to a Bitcoin transaction is made by adding a zero-value output with a *scriptPubKey* consisting of "6A σ δ", where σ is the size in bytes of the data to attach to the transaction, and δ is the data itself. Both values are encoded in hexadecimal notation.

```
6A 20 9D0F5692F0A7CCDBE5554732094CFB52
      589F5D8AD762BB54B77A1978462F01C2
```

Figure 5: *Sample scriptPubKey for anchoring a SHA-256 hash. Spaces and newlines must be ignored.*

A relatively high fee must be paid in order to rest assured that a transaction gets successfully accepted and confirmed by the Bitcoin network in a reasonable time even under heavy load circumstances. Recommended values are between 60,000 and 120,000 satoshi.

ii.   Ethereum anchoring

The Ethereum protocol provides a convenient method for anchoring hashes or data into transactions. This is done by creating a *message call transaction T* in which the data attribute T_d is set to be the Merkle root.

The protocol expects T_d to be an unlimited size byte array [13], so again the hashes resulting from the recommended hashing algorithm in this document—SHA-2 in its 256 bit version—fit perfectly into a single transaction.

A gas[1] value of 24000 szabo at a 20 Gigawei gas price is typically needed for broadcasting a message call transaction containing a SHA-256 hash.

iii.   Prefixing

Following Bitcoin community's convention for prefixing OP_RETURN operations in order to identify the generating application, a prefix should be added in front of the data itself, no matter which blockchain we are anchoring to.

At the time of this writing, there is no standarized format for prefixes, so different BTA implementations may choose to use it in a different manner.

By way of illustration, Stampery's BTA implementation uses the prefix format "53 υ ϕ", where 53 represents the letter "S"—standing for "Stampery"—in ASCII format, υ is one byte telling the BTA version again in ASCII, and ϕ is a two-bytes string identifying the server that broadcasted the transaction.

---

[1]Gas is the pricing or fee for running a transaction or contract in Ethereum.

```
6A 24 53 35 9B73
        9D0F5692F0A7CCDBE5554732094CFB52
        589F5D8AD762BB54B77A1978462F01C2
```

Figure 6: *Same scriptPubKey used in **Figure 5**, this time with proper prefixing and values $v = 5$, $\phi = $ "9B73". Spaces and newlines must be ignored.*

The ϕ value may be calculated by taking the first two bytes from the MD5 hash of the server's host name.

IV. Proof of anchorage

As mentioned before, the *binary hash chain* [9] that proves that a certain hash belongs to a tree is called *Merkle proof* or simply *proof*.

The proof for a single leaf hash contains the sibling hashes that need to be sequentially concatenated and rehashed in pairs in order to reconstruct the path from the leaf to the root.

$$\Pi(\alpha) = \{\beta, \ \gamma\delta, \ \epsilon\} \quad \Pi(\gamma) = \{\delta, \ \alpha\beta, \ \epsilon\}$$
$$\Pi(\beta) = \{\alpha, \ \gamma\delta, \ \epsilon\} \quad \Pi(\epsilon) = \{\alpha\beta\gamma\delta\}$$
$$\Pi(\delta) = \{\gamma, \ \alpha\beta, \ \epsilon\}$$

Figure 7: *Example illustrating the proofs (Π) for all the hashes in the Merkle tree from **Figure 4**. As it is noticeable, two contiguous hashes will likely have similar proofs, with the only one or two first items in the hash chain being different.*

In order to allow independent individuals and systems to verify the proofs of anchorage, there are some other elements that need to be distributed along the hash chain and the Merkle index. Those are: (1) the hash of the original data piece, (2) a reference to the chain in which the root was embedded, and (3) a transaction identifier that unambiguously points to the transaction where the root was embedded.

i. Proof formatting for internal use

Stampery's approach to proof formatting is really minimalistic and concise. A BTA proof of anchorage consists of one array or list containing the following items:

- $\Pi_{[0]}$: BTA version (currently 6)
- $\Pi_{[1]}$: Merkle index
- $\Pi_{[2]}$: Merkle proof (siblings list/hash chain)
- $\Pi_{[3]}$: Merkle root
- $\Pi_{[4]}$: Anchor tuple

    - $\Pi_{[4][0]}$: Chain ID
    - $\Pi_{[4][1]}$: Prefix

   – $\Pi_{[4][2]}$: Transaction ID

Current BTA version is 6. Prior versions used a different proof format which is not compatible with the one proposed here.

The Merkle root $\Pi_{[2]}$ and all the hashes in the proofs list $\Pi_{[1][n]}$ should be printed using uppercase. On the contrary, the transaction ID can be printed using either uppercase or lowercase.

Merkle proof lists with zero siblings can be represented by an empty brackets [], empty quotation marks "" or a NULL value.

```
[
 6,
 2,
 [
  "EFA4BA8F7A66BC3B...D3C358038F5A9C27"
  "7DDE76C5E472C9AE...B7B9567DCB3E9551"
 ],
 "1FA2DF8A8ABEA78E...EDC4E25207D2F125",
 [
   1,
   "9B73"
   "84ba00d2cebbb4ee...fbb06a053e4fba10"
 ]
]
```

Figure 8: *A BTA proof of anchorage, represented using JSON format. All the hashes and the transaction ID have been shortened for fitting this document's layout.*

ii.   Chainpoint and OpenTimeStamps compatibility

BTA proofs are 100% compatible with Chainpoint[14], a similar technology developed by Tierion, Inc. This means that the proofs generated by Stampery's BTA platform and the Stampery API can be easily presented in the Chainpoint format, as shown in **Figure 9**.

Similarly, BTA proofs can be also presented in OpenTimeStamps[15] binary format thanks to the extensibility of such proof format and protocol.

iii.   Chain IDs

The *chain id* is an integer number representing which blockchain is the proof anchored to.

Positive values correspond to production chains—*livenets*—while negative values correspond to their test counterparts—*testnets*.

ID no. 0 is left unassigned forever and shall only be used for protocol testing purposes.

```
{
  "@context": "https://w3id.org/chainpoint/v2",
  "type": "ChainpointSHA256v2",
  "targetHash": "9D0F5692F0A7CCDB...B77A1978462F01C2",
  "proof": [
    {
      "left": "EFA4BA8F7A66BC3B...D3C358038F5A9C27"
    },
    {
      "right": "7DDE76C5E472C9AE...B7B9567DCB3E9551"
    }
  ],
  "anchors": [
    {
     "type": "BTCOpReturn",
     "sourceId": "84ba00d2cebbb4ee...fbb06a053e4fba10"
    }
  ]
}
```

Figure 9: *The same BTA proof from Figure 8, in Chainpoint v2 format.*

Table 2: *Livenet Chain IDs*

ID	Chain
1	Bitcoin livenet
2	Ethereum Classic Livenet
3	Ethereum Fork Livenet
4	Litecoin

V.   PROOFS VERIFICATION

Verification of Merkle proofs is performed by reconstructing the path between the original data hash and the Merkle root. This is done by applying the mixing function to the hash and the first sibling $\Pi_{[2][0]}$ in the proof, then applying the same mixer to the resulting hash and the next sibling, and continuing this process subsequently with $\Pi_{[2][n]}$ until there are no more hashes left to merge. The resulting *root candidate* hash ρ is expected to correspond to the Merkle root $\Pi_{[3]}$.

The order in which the hashes are concatenated before the hashing function is applied can be easily derived from the Merkle index by using this function:

In the event that the resulting hash and the Merkle root mismatched, either the proof or the original data hash were modified, corrupted or tampered with at some time after the anchoring took place. This can be used as a method for checking file integrity over time—time stamp the same file

Table 3: *Testnet Chain IDs*

ID	Chain
-1	Bitcoin livenet
-2	Ethereum Rinkeby testnet
-3	Ethereum Morden testnet
-4	Litecoin testnet

$$o = \left(\frac{\iota}{2^n}\right) \bmod 2$$

Figure 10: *Given a merkle index ι and merkle proof $\Pi_{[2]}$, this function tells the order of concatenation o for any element $\Pi_{[2][n]}$ belonging to Π. $o = 1$ means left side concatenation ($Pi_{[2][n]} + h$) and $o = 0$ means right side concatenation ($h + Pi_{[2][n]}$).*

over and over and keep a history of the data hashes and proofs: if the file gets modified, the hash will change, will not match the one related to the previous proof and the date of modification will become evident.

In order to verify the validity of the proof and the anchor, it is also necessary to check if the root candidate or the Merkle root match the data payload in the transaction. This can be done by searching the transaction ID $\Pi_{[4][1]}$ in a block explorer[2], reading the payload and then checking whether it contains the root as a substring—probably at the end because of prefixing.

VI. Progressive proofing

Hashing is a quite intensive operation in terms of CPU usage. When dealing with big Merkle trees—with hundreds, thousands or even millions of leaf nodes—closing the tree and finding a final Merkle root is not trivial. Specifically, For any number ν of child nodes, the number of hashing operations needed to find the root is $\nu - 1$.

In addition, calculating the proof for every leaf node once the tree is closed means a lot of time wasted traversing it and picking the siblings.

Those are the reasons why we recommend to build the Merkle tree and the proofs as the hashes come and are appended to the leaf level instead of doing it at the time of closing.

Taking this into account, the logic to be carried out every time a new leaf node is added is described in **Figure 11**.

VII. Anchoring to multiple chains

As explained in Section III, it is specially convenient to perform anchoring to multiple blockchains in order to leverage the best of each one's feature set and also to have auxiliary anchors just in case some day one blockchain is somehow hacked or corrupted and the anchoring transaction is reverted, tampered with or simply disappears.

---

[2]Some popular block explorers are Blockcypher for Bitcoin and Etherscan for Ethereum.

```
algorithm prover is
  input: original data hash η,
         merkle proof π
         merkle index ι
  output: merkle root ρ

  ρ ← η
  for n in π
    σ ← π[n]
    o ← (ι / pow(2, n)) % 2
    if  o = 1 do
      ρ = mixer(ρ, σ)
    else do
      ρ = mixer(σ, ρ)
  return ρ
```

Figure 11: *An iterative proof verification function, described in pseudocode. The mixer function is the one described in **Figure 3**.*

$$\eta = \Pi_{[2]}$$
$$\eta \in \delta$$

Figure 12: *The candidate root η should match and be a substring of the data payload δ found in the anchoring transaction.*

i.   Parallel anchorage

Parallel anchorage is the simplest way to anchor Merkle roots to multiple chains.

It involves closing the Merkle tree every few minutes, calculating the root and then embedding it into multiple blockchains at the same time.

This way, the proofs for all the chains are identical with the exception of their last element—the anchor tuple $\Pi_{[4]}$—that will have different values for the chain ID $\Pi_{[3][0]}$ and transaction ID $\Pi_{[3][2]}$.

ii.   Incremental anchorage

Incremental anchorage is a more complex but also more powerful way to anchor Merkle roots to multiple chains.

It involves creating several different types of Merkle trees, each type corresponding to a chain we want to anchor to. Every of them will have its own *lifetime*—time before tree closing—based on the average block time of the matching chain.

For example, if we were to perform incremental anchorage with Ethereum and Bitcoin, we could define the following tree types and tree lifetimes:

A incremental anchorage procedure for those chains and lifetimes could be the following:

```
algorithm pusher is
  input: merkle tree τ,
         proofs list π,
         new hash η,
         level λ (default = 0)
  output: merkle tree τ,
          proofs list π


  if η ∉ τ[0] do
    σ ← τ[λ] or []
    σ ← σ + [η]

    if λ = 0 do
      π[η] ← []

    if mod(|τ[λ]|) = 1 do
      ς ← τ[λ][−1]
      ϖ ← mixer(ς, η)
      π ← proofer(τ, π, η, ς, λ)
      {τ, π} ← pusher(τ, π, ϖ, λ + 1)

  return {τ, π}
```

Figure 13: *A recursive hash appending routine, described in pseudocode. The mixer function is the one in* ***Figure 3***, *and the proofer function is described in* ***Figure 11***.

Chain	Tree lifetime
Ethereum	1 minute
Bitcoin	10 minutes

1. Create one tree for Ethereum and one for Bitcoin.

2. Receive all the hashes as they come and push them as leafs into Ethereum tree.

3. After 1 minute:

   (a) Close Ethereum tree and get its Merkle root.

   (b) Anchor the root to Ethereum blockchain.

   (c) Deliver the Ethereum proofs to the requesting parties.

   (d) Push the Ethereum root into the Bitcoin tree as if it were a leaf hash.

   (e) Create a new Ethereum tree for the next 1-minute frame.

4. Repeat steps 2 and 3 until minute 10.

```
algorithm proofer is
  input: merkle tree τ,
         proofs list π,
         left hash α,
         right hash β,
         level λ (default = 0)
  output: proofs list π
```

$$\phi \; \leftarrow \; \tau[0]$$
$$\kappa \; \leftarrow \; \tau[\lambda]$$
$$\rho \; \leftarrow \; \lfloor 2\hat{}(\lambda \; + \; 1)\rceil$$
$$\sigma \; \leftarrow \; \lfloor -\rho \; + \; |\kappa| \; \times \; (2\hat{}\lambda)\rceil$$
$$\epsilon \; \leftarrow \; min((\sigma \; + \; \rho), \; |\phi|) \; - \; 1$$
$$\iota \; \leftarrow \; \sigma$$

```
  for each hash (χ) in φ ∈ [σ, ε] do
    if ι < ⌊ρ / 2⌉ do
      π[χ] ← π[χ] + [α]
    else do
      π[χ] ← π]χ] + [β]
    ι ← ι + 1


  return π
```

Figure 14: *Auxiliar function for progressive proofing used by the pusher function in* **Figure 11**.

5. When minute 10 comes:

   (a) Close Bitcoin tree and get its Merkle root.

   (b) Anchor the root to Bitcoin blockchain.

   (c) Append each Bitcoin proof's siblings list $\Pi_{[2]}$ (*proof head*) to the end of all the Ethereum proofs' siblings list $\Pi'_{[2]}$ (*proof tail*) having a root $\Pi'_{[3]}$ that matches the original data hash from the Bitcoin proof.

   (d) Deliver the whole "merged" proofs to the requesting parties.

   (e) Delete all the old Ethereum trees and create a new Bitcoin tree for the next 10-minutes frame.

6. Start over the whole process.

Because of the way the Merkle proofs are generated and merged together, the resulting Ethereum proof will be a subset of the Bitcoin proof, but both of them will still be perfectly valid separately.

VIII. Distributed BTA (DBTA)

It is easy to imagine that a computer handling a Merkle tree with thousands or millions of nodes in its leaf level will need a huge amount of RAM memory. Furthermore, if the computer runs out of memory while building a Merkle tree, it will never be able to close it and all the hashes will be simply lost.

This kind of bottlenecks and situations can be easily overcome by using a Distributed BTA implementation as described hereafter.

DBTA is a computation cluster formed by a number ν of BTA instances called *nodes*.

A messaging queue system is used to distribute messages and balance workload between all the nodes.

In addition, a P2P protocol in conjunction to a basic consensus algorithm are used to reduce the number of Bitcoin transactions made by the cluster.

i. The cluster

We analyzed many different technologies and platforms when we built Stampery's own DBTA cluster, and finally opted for Erlang/OTP.

Erlang[16] is a programming language and virtual machine used to build massively scalable soft real-time systems with requirements on high availability. Some of its uses are in telecoms, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance.

OTP is a set of Erlang libraries and design principles providing middle-ware to develop these systems. It includes its own distributed database, applications to interface towards other languages, debugging and release handling tools.

For convenience, we wrote our BTA implementation using Elixir[17], a dynamic, functional language designed for building scalable and maintainable applications.

ii. The messaging queue

The messaging queue will fulfill a variety purposes, most of them related to the incremental anchoring procedure as described by Section VII.ii:

- Load balancing the distribution of incoming hashes to the nodes.
- Keeping the hashes in memory until they have been successfully anchored to all chains and the corresponding proofs have been delivered.
- Routing the Ethereum proofs back to their requesting parties.
- Load balancing the distribution of Ethereum roots from one node to another, so that they all receive an even amount of Ethereum roots to use as leaf hashes in their Bitcoin trees.
- Routing the Bitcoin proof heads to the nodes holding the matching proof tails.
- Routing the whole "merged" proofs to the requesting parties.
- Re-delivering hashes and roots to different nodes in case the ones that were assigned originally fail to perform their functions.

Therefore, the messaging queue system that we need must fulfill all those load balancing, routing and re-delivering requirements.

That is the case with RabbitMQ, the most notable implementation of the AMQP protocol[18]. We have been successfully using it in Stampery for DBTA since 2015 and it has proven to be the perfect fit for the set of features previously described.

To implement DBTA in RabbitMQ, the following queues are needed:

Queue name
eth
btc

In addition, it will be necessary to define the following exchanges[3]:

Exchange name	Type
eth	direct
btc	direct
proofs	direct

When a hash is received by a DBTA implementation, it must be published to the ETH exchange, which will route it to a different DBTA node every time[4]. At the same time, a binding will be created in the proofs exchange, using the hash as the routing key and pointing to a user-specific proof delivery queue, named in the form of a user identifier plus the $-clnt$ suffix.

The node will then push the hash into its latest Ethereum tree and will wait 1 minute before closing it. During that time, it will predictably receive approximately $\frac{\eta}{\nu}$ hashes, where η is the number of hashes sent to the exchange during the 1-minute frame, and ν is the number of DBTA nodes consuming the ETH queue.

Once the 1-minute frame is over, all nodes will close their Ethereum trees, get their roots and publish them to the BTC exchange. Such exchange will then route every root to a different DBTA node. At the same time, another binding will be created in the proofs exchange, using the root as the routing key and pointing to the same user-specific proof delivery queue, named as already stated.

The nodes receiving the roots from the BTC exchange will treat them as if they were hashes, push them into their Bitcoin trees and wait for 10 minutes before closing the trees. During that time, each node will predictably receive up to a maximum of 10 roots.

Once the 10-minutes frame is over, all nodes must close their Bitcoin trees and get their final Merkle root.

One important advantage of using AMQP is that if for some reason one or more nodes go offline or gets isolated from the rest of the architecture—in case of network failure, system maintenance, manual reboot, etc–it will collect the unacknowledged messages previously sent to such node(s) and reassign them to a different set of live nodes.

---

[3]Exchanges are AMQP entities where messages are sent. Exchanges take a message and route it into zero or more queues. The routing algorithm used depends on the exchange type and rules called bindings.

[4]A direct exchange delivers messages to queues based on the message routing key. Direct exchanges are often used to distribute tasks between multiple workers (instances of the same application) in a round robin manner.)

iii. Cluster leadership

At this point, if every DBTA node were to broadcast its own Bitcoin transaction, we would be incurring in a daily cost of up to $24 \times 6 \times \nu \times \beta$ USD[5], where ν is the number of nodes in the DBTA cluster, and β is the price of a Bitcoin transaction in US Dollars.

Because we are focusing on scalability and cost-effectiveness, we have figured out a way to keep the number of Bitcoin transactions steady so it is always the same regardless of the number of nodes in the DBTA cluster.

The trick consists of choosing a random *leader* node for every 10-minutes frame and letting it join all the roots into one and make a single transaction.

In Stampery's DBTA, this is made possible by Erlang's bundled distributed computing system[19].

All nodes must share the same magic cookie[6], so that they will be able to connect to each other automatically as soon as their EPMD[7] is set to discover nodes in the right domain names.

All nodes must have their clocks synchronized with one another with a maximum lead or lag that needs to be less than a half the duration of the tree lifetime corresponding to the fastest blockchain being used. This amount of time is called *consensus time*.

The time frames are fixed in time so that they can be synchronized across all nodes without need for signaling. For example, the Ethereum 1-minute frames will start at 00:01:00, 00:02:00, 00:03:00, etc.; while Bitcoin 10-minutes frames will take place at 00:10:00, 00:20:00, 00:30:00 and so on.

Every time a frame expires, the nodes make a list of their peers. Then they take the peers' names, append the Epoch time[8] for the starting second of the just finished frame, and hash the resulting string. Finally, the list of hashes is sorted alphabetically and the first one in the list is considered to be the leader.

Once the leader node has been chosen, the other nodes will report their Bitcoin roots to it, which will then build a final Merkle tree joining its own Bitcoin root and all the ones from its peers. After consensus time, it should close the tree, get the final Merkle root and anchor it to the Bitcoin blockchain.

The last thing the leader needs to do is reporting the Bitcoin transaction ID and the final proof heads to each of its peers, so that they can join them to their own Bitcoin proofs and report their proof heads back to the nodes holding the proof tails.

If the leader node failed to fulfill its duty in twice consensus time, the runner-up node in the list will take over and be the new leader for the unconfirmed frame. If it failed too, it will be the turn of the third node in the list, and so forth.

iv. Proofs delivery

When an Ethereum proof is ready to be delivered to the requesting party, it just needs to be published to the PROOFS exchange using the original data hash as the routing key.

---

[5]At the time of the last review of this document, this corresponds to $\nu \times 225$ USD per day.

[6]When an Erlang node tries to connect to another node, the magic cookies are compared. If they do not match, the connected node rejects the connection.

[7]Erlang Port Mapping Daemon

[8]Epoch time is a system for describing instants in time, defined as the number of seconds that have elapsed since 00:00:00 (UTC), Thursday, 1 January 1970, not counting leap seconds.

Likewise, when a Bitcoin proof is ready to be delivered to the requesting party, it just needs to be published to the same PROOFS exchange, this time using the Ethereum root as the routing key.

Due to the way in which AMQP direct exchanges work, both the Ethereum and Bitcoin proofs will be routed into the right proof delivery queue thanks to the routing keys and the previously created bindings.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009. `https://bitcoin.org/bitcoin.pdf`.

[2] Wikipedia, "Opcode — wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Opcode&oldid=728968013`.

[3] Bitcoin Wiki, "Op_return — bitcoin wiki." `https://en.bitcoin.it/w/index.php?title=OP_RETURN&oldid=60872`.

[4] M. Aráoz, "What is proof of existence?," 2014. `https://proofofexistence.com/about`.

[5] Stampery, "21st century notarization," 2014. `https://stampery.com`.

[6] Tierion, "Blockchain proof engine," 2014. `https://tierion.com`.

[7] Wikipedia, "Merkle tree — wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Merkle_tree&oldid=734149217`.

[8] R. C. Merkle, *A Digital Signature Based on a Conventional Encryption Function*. CRYPTO '87. Lecture Notes in Computer Science. 293. p. 396.

[9] Wikipedia, "Hash chain — wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Hash_chain&oldid=731737984`.

[10] U.S. National Institute of Standards and Technology, "Sha-3 standard: Permutation-based hash and extendable-output functions." Federal Information Processing Standards Publication no. 202, 2015. `https://dx.doi.org/10.6028/NIST.FIPS.202`.

[11] Wikipedia, "Length extension attack — wikipedia, the free encyclopedia." `https://en.wikipedia.org/w/index.php?title=Length_extension_attack&oldid=733680211`.

[12] F. Charlon, "Change the default maximum op_return size to 80 bytes." `https://github.com/bitcoin/bitcoin/pull/5286`.

[13] Dr. Gavin Wood, "Ethereum: A secure decentralised generalised transaction ledger - homestead revision," 2016. `http://gavwood.com/paper.pdf`.

[14] Wayne Vaughan, Jason Bukowski, Shawn Wilkinson, "Chainpoint: A scalable protocol for anchoring data in the blockchain and generating blockchain receipts.," 2016. `https://tierion.com/chainpoint`.

[15] Peter Todd, Ricardo Casatta, Luca Vaccaro, Andrew Poelstra, et. al., "Open timestamps: A timestamping proof standard.," 2017. `https://opentimestamps.org/`.

[16] Joe Armstrong, Bjarne Däcker, Thomas Lindgren, Håkan Millroth, et al., "Open-source erlang - white paper," 2000. `http://www1.erlang.org/white_paper.html`.

[17] J. Valim, "Elixir," 2012. `http://elixir-lang.org/`.

[18] Sanjay Aiyagari, Matthew Arrot, Mark Atwell, Jason Brome, Alan Conway, Robert Greig, Pieter Hintjens, John O'Hara, Martin Ritchie, Shahrokh Sadjadi, Rafael Schloming, Steven Shaw, Gordon Sim, Martin Sustrik, Carl Trieloff, Kim van der Riet and Steve Vinoski, "Amqp advanced message queuing protocol - protocol specification," 2006. `https://www.rabbitmq.com/resources/specs/amqp0-9.pdf`.

[19] Ericcson AB, "Distributed erlang," 2003. `http://erlang.org/doc/reference_manual/distributed.html`.