Book of Snowblossom

Joseph Gleason / Fireduck
v2021.09.13

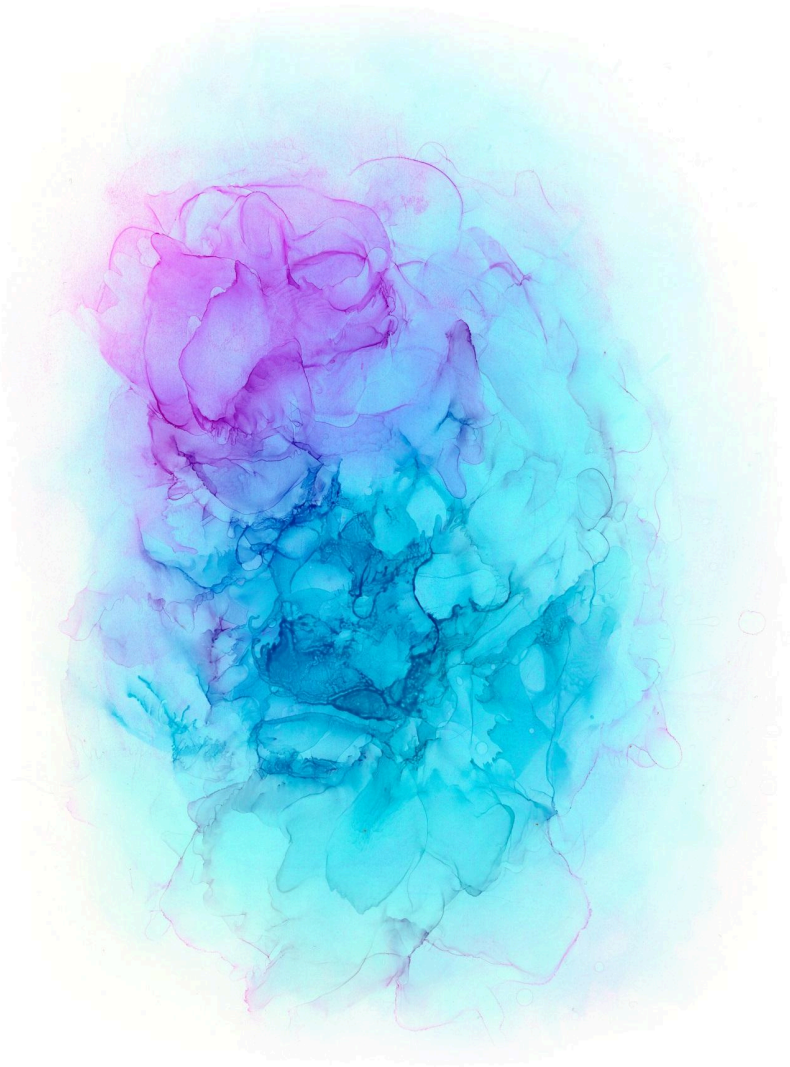

Table of Contents

Background

In 2012, I created the backend transaction processing and the provably fair process for Satoshidice.
In 2013, I created a Bitcoin mining pool implementation, SockThing. And of course, a mining pool, hhtt.
In 2014, I started on a java replacement for the then underperforming reference electrum server, jelectrum.

With these projects I got a very good idea of the internals of Bitcoin and the rough edges. Some key takeaways:

- Why are things so weird with byte ordering? (or maybe I am just dumb)
- Why are there so many ways to construct a payment to an address?
- This C++ code is pretty unreadable

Objectives

My objective with Snowblossom was to make a pure cryptocurrency that was simpler and had a cleaner code base.

Improvements

Snowblossom Braid (Sharding)

Overview

In general cryptocurrencies operate with a single sequential blockchain. Validation can only be done with a full understanding of the state of the coin (especially the UTXO set) and sequentially. This mostly limits any cryptocurrency to a small enough set of data to fit on one computer and the transactions rate low enough to be processed by one computer. Even if you tried to use multiple computers, the sequential nature of the blockchain makes the task of validation un-paralellizable.

Snowblossom Braid is an upgrade to make a set of interrelated sequential blockchains such that the work can be fragmented to be processed by multiple computers. This will allow for a much higher transaction rate at the cost of some overhead. This is commonly called sharding.

Important Cryptocurrency Behaviors

In discussing any sharding scheme, it is important to highlight behaviors of single chain cryptocurrencies that users depend on to discuss how the sharding solution maintains those behaviors.

Finality of Transactions

Users seem to be mostly on board with the concept that a transaction is first pending or at-risk and then later it will be confirmed and then more confirmations will pile on in subsequent blocks until the user is satisfied that the transaction will not be reversed (by a block chain reorg) and can be counted on.

With this Snowblossom Braid this is all still true, but the finality evaluation is a bit more complex. Rather than just asking how many confirmations are on a transaction, we have to ask what shards have included the block with the transaction. There will be some client work to make this clear to the user.

Fire-and-Forget

When a user sends a transaction on a peer to peer cryptocurrency network and the transaction is accepted into the mempool, the user can be fairly confident that the transaction will eventually be confirmed. They don't have to do anything else. Of course, it isn't a guarantee until the

transaction is confirmed but in most cases the user can simply send the transaction and then disconnect from the network and go plant some flowers.

This behavior is maintained with the Snowblossom Braid work with one exception. In the braid system, a user might have unspent funds on multiple shards. This would mean if they want to send and don't have enough on one shard they will have to do (really their client software will do this) multiple transactions for the desired send. This could be done in two ways:

1) Send partial payment from each shard as needed. This way, the recipient gets the funds as quickly as possible and fire-and-forget works fine.
2) The user could do one transaction to consolidate funds onto one shard and then later send the transaction to send to the destination. This would not work with fire-and-forget without some extra work, like a special mempool for transactions waiting for inputs to become available.

No Double Spends / Consistent State View

This is related to the above fire-and-forget but more on the recipient side. When you see a pending incoming transaction, you want to be able to know that it is very likely that the transaction will confirm. You should be at least confident enough to let the payer leave with a coffee or a toaster but perhaps not with a car. For a car, you might want to wait for some confirmations.

Part of this is done by the standard First Seen, First Added behavior. This means the first valid transaction that spends a particular transaction output gets to claim it and no other transactions spending that output will be accepted into the mempool. This isn't a guarantee as different nodes might have a different view of the state of the mempool or nodes might be restarted and lose the mempool state. However, most of the time, this works fine and can be relied upon for small transactions. This continues to work with Snowblossom Braid since each output is bound to a shard at creation time (when the transaction is made) so the normal mempool rules apply for transactions spending it from that shard. No other shards could spend it.

Users Don't Care About the Details

While a user should always be given the option to inspect the details, for the most part they don't care. They don't want to do UTXO management. They don't care what shards their outputs are on and they shouldn't have to know. While it hasn't been completed yet, the user facing client code for Snowblossom Braid should take this into account.

Development and Deployment

Snowblossom Braid is operational on testnet from a branch of snowblossom git named ‘shardo’. It has been heavily tested with networks up to 128 shards and the code works well.

Soon we will have a discussion about bringing it into mainnet.

If it does go into the mainnet, the total block reward will be the same. It is more complicated, but the end sum will be the same. Rather than a simple block reward of 50 SNOW for a block, it will be 50 SNOW spread among all the then existing shards for that block height. For even more fun complexity part of the reward is for including blocks from other shards. But it will still sum to the same reward per block height as the single shard chain. So the coin supply will remain the same.

Shards are triggered by usage so initially mainnet would just be a single shard (as it is now). Also, all existing history, transactions and addresses will be maintained. This would be an in place protocol upgrade, not a new coin.

Structure

The braid structure is called a braid because like a hair braid, there are strands that are in some ways separate but also linked to each to form one unit. It is a set of interwoven blockchains. Each shard will have a sequence of blocks like any other blockchain with the following changes:

- Each block may include headers of blocks from other shards.
- A transaction may only spend from inputs in the current shard. They may write outputs to any other shards. The outputs are marked as part of the transaction data for which shard they can be spent on by the transaction creator.
- Each shard can only get so far ahead of other shards, this will be a defined max distance. For example, if we set the max distance to 4, then in order for a shard to make a valid block of height 1000, it must include headers for other shards up to at least height 996.
- The included headers for other shards must form a sequential block chain. This means if we include the header for the block on another chain, we must have already included the parent of that block. This is how we manage the UTXO handover, by making sure we get each block from the other chains in order to import the relevant UTXOs.

UTXO Management

When a block is created on a shard, in addition to its own internal UTXO management, any transaction outputs that go to other shards form an export set. These are UTXOs that need to be encoded into other shards when they include this block.

When a block is included, the export set from that block to this shard is integrated into the shard's UTXO.

This way, we have a coherent way to move UTXOs from one shard to another safely. The source shard includes the output exactly once. The UTXO is imported in the target shard only when they include the block that has it.

Shard IDs and Creation

Each shard has its own PoW difficulty based only on the block rate of the shard itself.

Rather than deciding a number of shards up front, since each shard has some additional overhead we decided to make shards on demand and only define a max number of shards in the protocol. To make this work each shard has a running transaction size value that indicates how full recent blocks have been. When the shard is over the protocol defined threshold of fullness the shard will fork. This shard will stop getting new blocks and two new shards will start using the last block of the old shard as a parent. The new shards will have half the PoW difficulty, half the block reward of the parent shard. One of the shards will inherit the parent's UTXO. The other will start with an empty UTXO. The shard that inherits the parent's UTXO will also import any UTXO exports to the parent shard ID.

Example, when shard 2 splits it creates shards 5 and 6. Shard 5 will inherit the UTXO from shard 2. Any future exports to shard 2 will be imported by shard 5.

In addition, a shard will import the UTXOs to any future child shards. For example, if before shard 2 splits, there is an output to shard 5 it would be imported by shard 2.

This means that once sharding is enabled on the network, transactions will be able to write outputs to any valid shard ID. Those will be mapped to currently existing shards.

The Shard ID structure is defined in ShardUtil. In general the form is, the children of shard N are:
N*2+1 (inherits UTXO) and N*2+2

Here is the first few layers of the tree:

```
          0
     1          2
  3    4     5     6
7 8 9 10 11 12 13 14
```

Note: as the shards decide to split based on their own internal loading, they might not expand at the same time. For example, the braid might consist of shards: {1,5,6} or {1,2} or {3,4,5,6}

Block Rewards

Block rewards are a bit more complicated than the single blockchain case while still summing to the same number.

The block reward for a block in a shard is:
Let shard_faction be the fraction of the total tree this shard represents. For example, shard 0 would be 1/1 (the entire tree). Shard 1 would be (½). Shard 5 would be (¼).
Let direct_faction = 0.75
Let indirect_faction = 0.25
Block reward = general_block_reward * shard_faction * direct_faction
+ general_block_reward * shard_faction * indirect_faction * shard_faction
+ shard_faction * sum(general_block_reward * indirect_faction * included_shard_faction)

In other words, a slice for the block itself, then a slice for each block from another shard we include. This creates an incentive to include other shards as often as possible, making a tightly linked braid.

Here it is in code: ShardUtil.getBlockReward

Scenarios

Bad Block

This is pretty much the worst case scenario. It is nasty but if miners are careful (and it makes them more money to be careful) it will never happen.

Suppose there are 6 shards, A,B,C,D,E,F. A miner creates a block on A at height 100 (We'll notate this as A100). A100 has a valid header but there is an invalid transaction. The other miners get the header for A100, the header validates so they use it (miners shouldn't do this, they risk having a lot of orphaned blocks if they include blocks they don't fully validate). But

they do. So the shards B-F all get new blocks built including A100. But the whole block for A100 doesn't validate so no miners make an A101 because they can't. The rest of the network doesn't care, and we get B105 through F105 all made. At this point, they can't go to height 106 because A100 is too old, they can't make any more blocks without A101.

So any mining on B-F stops. They have no useful work to do. Miners can't make A101 off A100 because it is invalid. So miners make a new valid A100. However, B-F 100-106 all include the bad A100. The miners don't want to orphan their own blocks so they drag their feet trying to make any new B-F blocks until A gets to A105 and can't continue without making more blocks on the other shards. So they eventually start making new B100-F100, reorging all those shards and the network goes on.

This is terrible, but the network will reorg and move on. Hopefully miners won't be stupid enough to include blocks they haven't validated themselves but even if they do, it will work out.

Reorg an shard

A concern with PoW based cryptocurrencies is the so-called 51% attack problem. The problem is simply put, an entity that controls over half the mining power can rewrite the blockchain as far back as they like. So with a sharded system, is it that much easier to rewrite a single shard? Imagine there are 10 shards with rough even PoW difficulty. An attacker with only 5% of the hash power could rewrite a shard, since only about 10% of the network PoW is on any given shard, right? Not in the case of the Snowblossom Braid. If an attacker recreates some blocks on a shard, the miners will ignore it since the pre-existing blocks are already braided into other shards. They would make more money by not orphaning their own blocks to follow some re-org fork of a shard, even if that re-org seems like the current best for the attacked shard.
And since miners make more money by including other shards in their blocks, the shards will almost always be tightly interwoven.

Trust / Confirmations

In traditional cryptocurrencies faith in a transaction is based on confirmations - how many blocks deep is a transaction.

The concept remains the same, but phrased: how many blocks have been mined that would have to be discarded to remove a transaction.
For example, let's say a transaction is included in a shard block and then that block is included in two other shards. All three of those blocks would have to be orphaned for that transaction to not take place. So even though the transaction is only one confirmation deep in its own shard, by taking a holistic look it could be considered 3 confirmations. However, in a traditional

cryptocurrency 3 confirmations means three blocks at the total network hash rate. While three shards is just the hash rate of those three shards.

Maybe confirmations will become a float. So when half the shards include the block with your transaction you have 0.5 confirmations. When all the shards have and some of them have two blocks on it, then 1.2 confirmations.

Deep Block Proof

Suppose there is some node A that is a full validator for shard S, meaning it has ingested and checked all blocks.

Suppose there is some node B that is not a full validator of shard S. It is only looking at and storing headers.

Supposed B accepts that block N on shard S is valid, due to network concensus.

A can make a proof that proves that block N+1 is valid to B.

This can be done by A providing the block N+1. In addition A would provide enough of the UTXO internal nodes to prove that all Transactions Outputs spend by that block were in the block N UTXO hash. A would also provide other internal nodes to prove that the UTXO changes in block N+1 mutate to be the UTXO hash in block N+1.

This would be a significant chunk of the UTXO tree, but not nearly all of it.

B, using this data to validate the block could then discard rather than store the validation data.

This would be some intense code, but it is very doable.

Ecosystem

I suspect most mining pools that want to be competitive will run validation nodes on all shards. That way, they can mine on any shard if it makes sense to do so. Also they can validate blocks on all shards to be able to include the most valid other blocks into blocks they mine to make the most block reward.

Miners who don't have that much hardware may collaborate with trusted peers for block validation or use a third party validation service.

I could also see a third party service existing for address lookup, since a wallet software won't necessarily know which shards might contain an output for their addresses.

The Dance

As I program this braid and run into many problems forming braids. Due to duplicate blocks with some shards following one chain and others following others it ends up in states that are hard to make progress and the chain stalls. It should be noted that progress is always possible, even if it must orphan some blocks, but that doesn't mean finding the path forward is easy.

Anyways, the dance is a way to side step this problem until someone smarter than I can solve it. The dance will *not* be encoded in the validation rules, a node may build blocks not following it without breaking the protocol, they just risk orphaning if other nodes are insisting on following the dance for block creation. This way, if some folks figure out a better way it isn't a break change to improve the network.

Anyways, with the dance, the shard that currently inherits utxos for shard 0 shall be the coordinator.

When making new blocks, the coordinator shard may include any blocks from other shards that follow the dance (and all other network rules).

Non-coordinators may only include blocks from:

- The coordinator shard
- Other blocks already included by the coordinator shard

This way, the code is much simpler. Rather than mucking around with gold sets and trying to find solutions to intractable problems, the coordinator simply:

- Looks for blocks that extend from existing included blocks as long as they follow the dance and includes them

For non-coordinator blocks they simple:

- Include the latest known coordinator shard blocks and any blocks included by the coordinator blocks.

The downside is as follows: suppose there are four shards: {3,4,5,6}. Shard 3 will be the coordinator shard. Lets say shard 6 is exporting some utxos for shard 5. In order for those UTXOs to be spendable:

- Shard 6 must mine a block
- Shard 3 must mine a block and include the new shard 6 block

- Shard 5 must mine a block including the new shard 3 block and the new shard 6 block.

Without the dance, only shards 6 and then 5 need to mine a block. With the dance, there have to be those three, in that order. This increases the time before transfer UTXOs are spendable but this seems like a reasonable compromise.

UTXO Improvements

UTXO indexed on address,txid,out_idx

In Snowblossom, the UTXO is indexed by recipient address, then transaction ID and finally output index in the transaction. This allows the UTXO trie to be used to quickly generate UTXO proofs for light clients. The server can send along trie nodes needed to prove the completeness or lack of UTXO for any given address. To make this work, there needs to be a uniform way to express the addresses and be able to get the exact address for any transaction input or output. This is an advantage of the AddressSpec model as opposed to the Bitcoin OP-code approach which can express addresses in different ways.

UTXO root hash in block header

As the UTXO root hash is a key component in the blockchain, it makes sense to include it in the block header. This way, light client UTXO proofs can be linked directly back to the block headers.

UTXO stored in hashed trie

A difficulty of storing the UTXO root hash in the blockchain is you need an absolutely consistent way of expressing and storing it. The normal way is using a trie, a tree structure with specific rules such that the same set of data will always have the same tree structure. This way, the tree can be hashed and produce a single hash on all nodes. However, in a blockchain that can have re-orgs this can be a database challenge. Traditionally to do a reorg, you would need to roll back removed blocks and then apply the new blocks. However, if you haven't validated the new block UTXO root hash yet, you don't want to make those sort of database changes. You can't be sure the new blocks are valid yet. So I have invented a new data structure, Hashed Trie. In this structure, each node in the trie is stored in an underlying key value store with the hash of the node (and all the child nodes) as the key value. Using this method, each UTXO root is stored and retained and the database can in fact track multiple block chains at once. Also, a client could query the UTXO of any previous block if they want to.

Hashed Trie

I think this is novel, but if I am wrong, please let me know.

Implementation - Hashed Trie Source Tests Snowblossom Trie Tests

Definition

A Hashed Trie is a structure that is a Trie where each node has a hash value based on its contents. In my implementation there is an underlying map of hashes to nodes: Map<Hash,Node>. The tree state is saved as a hash, which simply points to the root node for that state. The children of each node are referenced by their hash. No node is every overwritten (except as possibly rewritten as the exact same data), if it is changed the hash changes and it is saved under the new hash. This makes it a space efficient Copy-on-Write (CoW) structure.

Each operation is given the root hash as a parameter. Any modify operation returns a new root hash for the new tree.

The root hash represents the contents of the tree. Since there are fixed rules for the structure of the tree, the same keys and values will always result in the same root hash.

Advantages

- Space efficient. Even though we end up storing many variations of the same tree, we only duplicate the nodes that are different.
- Previous versions of the tree are readable as long as you have the previous root hash.
- Any mutation can be done from any previous root node. No need to ever back out any changes, just use a previous root hash.
- No need to lock for writes. Reads and writes can happen simultaneously.

Disadvantages

- Any write will involve rewriting all the nodes from the root on down to the leaf node in question log(n) operations. Can be helped by batching updates.
- Reads involve reading all the nodes along the path, log(n) operations (likely helped greatly with cache).
- Since each mutation involves taking the old root hash and returning a new root hash, if there are two writes that you want in the tree you need to do them in a batch or in sequence.
- All previous nodes are always kept. There is no pruning.

Why it is awesome for blockchains

- The root hash can be shared in consensus to insure that nodes have the same data in the tree. For example, in Snowblossom the UXTO root hash is from the UTXO hashed trie.

- Then the shared root hash and the intermediate nodes can be used to prove the existence and completeness of any data in the tree to light clients or header only nodes.
- As the structure has fixed rules, the proof can also prove that data isn't there. Example: by showing the parent node of where the data would be, if it existed.
- If there is a reorg, or potential reorg the new root hashes can be updated based on the root hashes of the previous blocks. No need to back out changes or pick a chain, just apply all reasonable blocks to the hashed trie. Use whatever ends up winning. For example, lets say the trie is storing the transactions that have been confirmed as of the most recent block. Lets say we are on block 10000. Suppose there is a reorg so a new block 9997 comes in. We build the transaction trie for that simply taking the root hash from block 9996 and mutating from there. Then the new chain fork can be imported independently of the existing fork.
- If we save the root hash for previous blocks, we can query the tree for the state of things at any previous block we want. This way, a client that is doing a long read of a bunch of stuff can pick and block and query everything relative to that even as new blocks are coming in.
- Since for a blockchain ledger, we generally don't want to throw data away, the fact that nothing is ever pruned is fine.

Wire Messages and Network Protocol

All Snowblossom messages and peer-to-peer network interactions are defined in protobuf and uses gRPC. This allows the key interactions of the system to be well defined and consistent.

proto files

Peer-to-peer communications (and light client connections) support TLS with actual certs. This is done without the hassle of having the certs issued by a certificate authority by having the key id of the server node being known by the client. The p2p network gossip sends the key IDs with the node gossip data. The hard coded seed nodes have the key IDs hard coded. This makes a MITM attack impossible without depending on any certificate authority.

The client checks the server cert to make sure it is signed by the expected signing key.

TLS source

StoatPOW - Storage based IO access PoW

Concept

The concept of this Proof of Work is that a variety of general computing use cases love fast access to large storage. Examples: gaming, database, media editing

So any advances in fast access to large storage will make its way into commodity parts relatively quickly and thus be generally available and useful. So the Snowblossom proof of work is based on fast access to large storage.

Snowfields and Difficulty

The snowfields are large deterministic data files. The smallest was 1 GB, the current field is 256 GB. The field size doubles as the hash rate goes up, every 4x increase in hash rate doubles the snowfield size. Once the field size is increased it never decreases.

With these large files, we don't have every node or client that is only verifying blocks to need to have access to them. So the merkle roots of the fields are hard coded. As part of the mining process the miners include proofs of the data segments they read from the snowfields that prove the data segments were correct, in the right locations and produce the right hash.

Snowfield Generation

Snow field generation is complicated because they must have the following properties:

- Deterministic - anyone should be able to regenerate them
- Non-parallelizable - it should be impossible build just part of the snow field on demand

Towards this end, the program to create the snowfields is called snowfall. It does multiple prng passes over the file. In essence, you can think of it as a pseudo random number generator with an absolutely huge state space. This prevents anyone from effectively checkpointing the generation state. We want to avoid a situation where a miner could generate parts of the snowfield on demand rather than needing to load the pages from storage.

Multiple signing algorithms

All addresses are multisig (1 of 1 in the common simple address case)

Client wallet format super safe

I am always suspicious of binary file formats that I can't easily inspect. Especially for critical things like cryptocurrency wallets. Fortunately, as a crypto wallet is a small data set and all operations can be considered append operations we can make some nice decisions to make them safer than the traditional "wallet.dat" approach. With a single file I am always afraid of things like if I open the wallet with a new software version, will I still be able to use older software? What if I accidentally cause new keys to be generated? Then I need to update my backups. What if I load the wallet on two different computers, how do I merge them? Is that even possible? The Snowblossom approach solves all this.

General Wallet Data Updates

In the Snowblossom CLI client as well as the IceLeaf GUI client the wallet data is saved in a directory rather than a single file. Each write is written as a new file with a randomly generated filename. On wallet load, all the files are read and merged in memory and then written out to a new combined file and only after that new file is written and flushed, the source files are removed.

Format Updates

The save files themselves are instances of the protobuf WalletDatabase. This allows fields to be added as features require them. There is a version field, if this is higher than what the source code WALLET_DB_VERSION then when the database files are merged, the original files are not removed. The assumption is there might be new fields that the current source code does not know how to correctly merge. So in the case of mixing old and new software versions, there is a possibility of database files building up but everything should work fine.

Export as JSON

In the Snowblossom CLI there is an export as json operation. It isn't pretty but it lets the user inspect what is going on.

Supported Use Cases

Given the above merge and file naming behavior, the options for wallet use are wide open. Things that work great:

- Realtime syncing between computers via NFS or shared file system or simply running multiple clients on one computer
- Eventual syncing like Dropbox or unison sort of things
- Merge wallets by just copying the files into one directory
- Mixing old and new snowblossom clients

Usual Features

- Block Time Average of 10 minutes
- Child-pays-for-parent (CPFP)
- Transaction immutability
- Double-spend protection
- Resilient peer-to-peer network
- Decentralized design
- Halfing-block reward over time
- First Seen First Added