Meeting notes 2019-01-29

A case against the half-blocks

Tribler's current Trustchain implementation uses half-blocks to enable concurrent transactions, to deal with slower peers. When only allowing one transaction at a time a delay in a single peer can prevent the creation of further blocks.

The half-blocks essentially are trade-proposals which are directly added to the proposer's chain. If the counter-party chooses to accept this trade, he can then create the other half of the block which will refference the first half.

Since the creation of the half-block doesn't depend on the counter-party there is no communication or negotiation overhead, resulting in a high throughput.

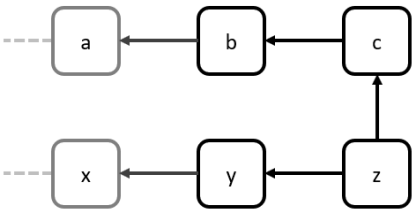

Fig 1. Graphical representation of two individual chains, linked together through a common transaction.

Problem of verification

While this seems to solve the problem of unresponsive or slow counter-parties, it only delays the outings of it. The question now is, how to deal with half-blocks while verifying a chain.

A trivial solution is to treat a chain with uncompleted half-blocks as invalid. However, this results in undoing all the benefits of the half-blocks, as one would only append a half-block if the other half is already signed (nobody would risk the change of an invalid chain). Thus, only a new transaction can be started if the previous is successfully completed. We require a way to handle invalid blocks without rendering the chain invalid.

When assessing the current state of the chain the half-blocks should be treated in a sensible way, even more so if we would want to store the state[1] in every block. There are two options: Include the transaction directly in the current state or wait until the other half is received.

Directly updating the current state:

> When including the transaction directly into the current state, the current state basically becomes worthless: Unless the complete chain is analyzed to determine which portion of transactions that are valid, there is no way of knowing the actual current state.
>
> *(This is the current implementation in Tribler with respect to the 'running total'.)*

[1] In the case of a simple cryptocurrency-like token the state would be the token balance of that user. Storing the state of the chain after each transaction in the block greatly improves verification performance, since an analysis of the whole chain isn't necessary to determine the current state.

Update after receiving the corresponding counterpart:

> When including the transaction in the next block after receiving a signed counterpart, transactions becomes difficult to verify, as the current state seems to update with a randomly seeming offset. Even when referencing the block that supposedly got fulfilled there's no way of knowing if that block ever existed and was part of the chain unless the verifier walks the chain until he finds it (or reaches the genesis block if it doesn't exist).
>
> The issue of seemingly random changes in the state could also be solved by implementing a new type of block, signed by both parties with ca copy of the transaction that is completed. This makes the state-change in a logically and easy to verify manner, however the counter-part is then redundant as the information is already stored in the block that signals the completion of the transaction.

In both cases it the benefit of being able to determine the state without verifying the complete chain is lost.

Life without half-blocks

The issues associated with the verification can be circumvented by not using half-blocks in the first place. Doing so might slow down the block creation but improves block verification as transaction can be checked in a single lookup and uncertainty is removed: Either the block is invalid and always will be or it is valid and always will be.

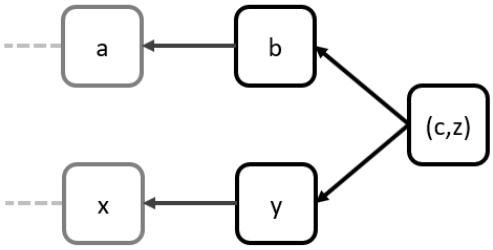

Fig 2. Graphical representation of a jointly created block.

When creating blocks there are two major sources of latency:

> Communication induced latency: *Latency due to messages/packet delayed or dropped.*
>
> Negotiation induced latency: *Latency which finds its origin in reaching agreement on the details of the trade.*

If no mutations to the chain is required while negotiating a trade, multiple trades can be negotiated in parallel, increasing the throughput of the network.

An argument against parallelization is that by preparing multiple transactions concurrently one is essentially creating temporary forks on his on timeline. Which opens up a whole variety of potential problems, look for example figure below:

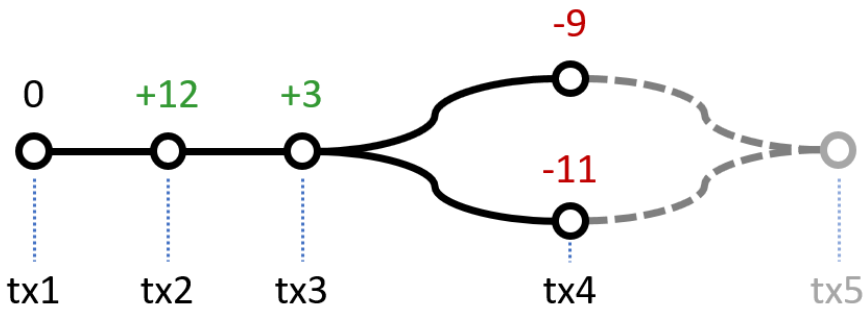

Fig 3. Schematic overview of transaction-timeline with a fork due to concurrency.

In figure 3 we have transactions of units seen from a single node's perspective. A positive number means a transaction in which the receive units, where negative values are transaction in which they lose units. This example assumes that one is not allowed to have a negative unit balance.

TX1 to TX3 pose no threat but problems start to arise at TX4. At the initiation of TX4 both parties might not know of the other party's transaction in progress and both seem valid transactions. The total sum of tokens becomes negative, which violates the requirements of always having a positive balance (i.e. a double spending attack).

While some solutions might be possible, it's all but trivial (Checking the chain after creating but before committing results in a race condition, and rendering one of them invalid is difficult; if transactions can be revoked later no node can ever trust another node on their balance)

Explicit forking

The above-mentioned problem lies mainly in the fact that every time concurrent transactions occur a temporary fork is created, but parties are unaware of the other forks. If concurrency is required, (for example due to slow negotiations) explicitly mentioning the fork will solve the problem.

When creating an explicit fork both parties sign a block indicating they're engaging in a transaction, this block also includes the maximum transaction volume associated with the fork. By doing so it is guaranteed that the value associated with the current fork cannot be spend in another branch of the chain. When the transaction is completed a transaction-block is created and the fork merges back with the rest of the chain.

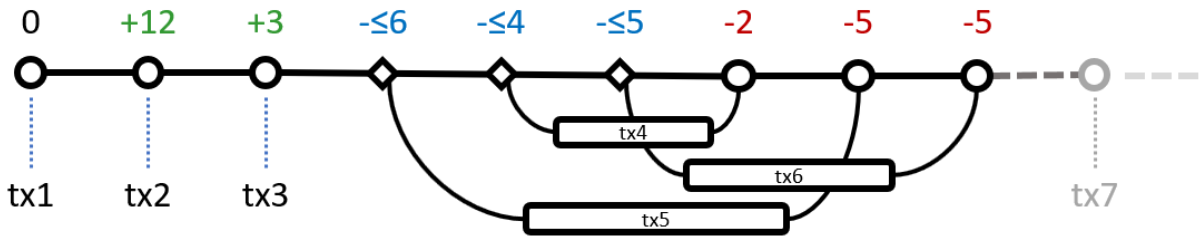

Fig 4. Schematic overview with explicit forking with three concurrent transactions.

The example in figure 4 uses the same concept as figure 3, however now it creates a block to explicitly indicate a fork in the timeline. Due to this tx4 and tx6 can be initiated while tx5 is still in progress. When initiating a new transaction, it becomes directly clear what the current state of the chain is.

The number of concurrent transactions can be limited by not allowing the oldest fork to be older than X blocks. This greatly increases the verifiability of a chain since the maximum number of blocks that need to be checked before ensuring there are no open forks = $X \times 2$.