Re: [Bitcoin-development] Tree-chains preliminary summary

Peter Todd Tue, 25 Mar 2014 05:39:15 -0700

On Sat, Mar 22, 2014 at 12:43:34PM -0700, Mark Friedenbach wrote:
> Btw, any chance we could get a summary description of tree-chains
> posted to bitcoin-development?

Advertising

Sure:

Introduction
============

Bitcoin doesn't scale. There's a lot of issues at hand here, but the most fundamental of them is that to create a block you need to update the state of the UTXO set, and the way Bitcoin is designed means that updating that state requires bandwidth equal to all the transaction volume to keep up with the changes to what set. Long story short, we get $O(n^2)$ scaling, which is just plain infeasible.

So let's split up the transaction volume so every individual miner only needs to keep up with some portion. In a rough sense that's what alt-coins do - all the tipping microtransactions on Doge never have to hit the Bitcoin blockchain for instance, reducing pressure on the latter. But moving value between chains is inconvenient; right now moving value requires trusted third parties. Two-way atomic chain
transfers does help here, but as recent discussions on the topic showed there's all sorts of edge cases with reorganizations that are tricky to handle; at worst they could lead to inflation.

So what's the underlying issue there? The chains are too independent. Even with merge-mining there's no real link between one chain and another with regard to the order of transactions. Secondly merge-mining suffers from 51% attacks if the chain being merge-mined doesn't have a majority of total hashing power... which kinda defeats the point if we're worried about miner scalability.

Blocks and the TXO set as a binary radix tree
=============================================

So how can we do better? Start with the "big picture" idea and take the linear blockchain and turn it into a tree:

```
  ┌───────┴───────┐
  ┌───┴───┐       ┌───┴───┐
  ┌─┴─┐   ┌─┴─┐   ┌─┴─┐   ┌─┴─┐
  ┌┴┐ ┌┴┐ ┌┴┐ ┌┴┐ ┌┴┐ ┌┴┐ ┌┴┐
  │  │  │  │  │  │  │  │  │
```

Obviously if we could somehow split up the UTXO set such that individual miners/full nodes only had to deal with subsets of this tree we could significantly reduce the bandwidth that any one miner would need to process. Every transaction output would get a unique identifier, say txoutid=H(txout) and we put those outputs in blocks appropriately.

We can't just wave a magic wand and say that every block has the above structure and all miners co-ordinate to generate all blocks in one go. Instead we'll do something akin to merge mining. Start with a linear blockchain with ten blocks. Arrows indicate hashing:

```
a0 ⇽ a1 ⇽ a2 ⇽ a3 ⇽ a4 ⇽ a5 ⇽ a6 ⇽ a7 ⇽ a8 ⇽ a9
```

The following data structure could be the block header in this scheme. We'll simplify things a bit and make up our own; obviously with some more effort the standard Satoshi structures can be used too:

```
struct BlockHeader:
  uint256 prevBlockHash
  uint256 blockContentsHash
  uint256 target
  uint256 nonce
  uint time

For now we'll say this is a pure-proof-of-publication chain, so our block contents are very simple:

```
struct BlockContents:
  uint256 merkleRoot
```

As usual the PoW is valid if H(blockHeader) < blockHeader.target. Every block creates new txouts, and the union of all such txouts is the txout
set. As shown previously(1) this basic proof-of-publication functionality is sufficient to build a crypto-currency even without actually validating the contents of the so-called transaction outputs.

The scalability of this sucks, so let's add two more chains below the root to start forming a tree. For fairness we'll only allow miners to either mine a, a+b, or a+c; attempting to mine a block with both the b and c chains simultaneously is not allowed.

```c
struct BlockContents:
  uint256 childBlockHash # may be null
  bool childSide # left or right
  uint256 merkleRoot
```

Furthermore we shard the TXO space by defining txoid = H(txout) and allowing any txout in chain a, and only txouts with LSB=0 in b, LSB=1 in c; the beginning of a binary radix tree. With some variance thrown in we get the following:

```
                b0  b1  b2  b3  b4  b5  b6  b7  b8
  a0  a1  a2  a3  a4  a5  a6  a7  a8
      / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \
  c0  c1  c2  c3  c4  c5  c6  c7
```

We now have three different versions of the TXO set: \(\Sigma a\), \(\Sigma a + \Sigma b\), and \(\Sigma a + \Sigma c\). Each of these versions is consistent in that for a given txoutid prefix we can achieve consensus over the contents of the TXO set. Of course, this definition is recursive:

```
                a0  a1  a2  a3  a4  a5  a6  a7  a8
  c0  c1  c2  c3  c4  c5  c6  c7
      / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \ / \
  d0  d1  d2  d3  d4  d5  d6
```

Unicode unfortunately lacks 3D box drawing at present, so I've only shown left-sided child chains.

Herding the child-chains

========================

If all we were doing was publishing data, this would suffice. But what if we want to synchronize our actions? For instance, we may want a new txout to only be published in one chain if the corresponding txout in another is marked spent. What we want is a reasonable rule for child-chains to be invalidated when their parents are invalidated so as to co-ordinate actions across distant child chains by relying on the existence of their parents.

We start by removing the per-chain difficulties, leaving only a single master proof-of-work target. Solutions less than target itself are considered valid in the root chain, less than the target \(<< 1\) in the
root's children, << 2 in the children's children etc. In children that
means the header no longer contains a time, nonce, or target; the values
in the root block header are used instead:

```c
struct ChildBlockHeader:
    uint256 prevChildBlockHash
    uint256 blockContentsHash
```

For a given chain we always choose the one with the most total work. But
to get our ordering primitive we'll add a second, somewhat brutal, rule:
Parent always wins.

We achieve this moving the child block header into the parent block
itself:

```c
struct BlockContents:
    ChildBlockHeader childHeader # may be null (zeroed out)
    bool childSide # left or right
    bytes txout
```

Let's look at how this works. We start with a parent and a child chain:

```
a0 ⇽ a1 ⇽ a2 ⇽ a3
  \                     \  
 b0 ⇽ b1 ⇽ b2 ⇽ b3 ⇽ b4 ⇽ b5
```

First there is the obvious scenario where the parent chain is
reorganized. Here our node learns of a2 ⇽ a3' ⇽ a4':

```
a0 ⇽ a1 ⇽ a2 ⇽ a3' ⇽ a4'
  \                     \  
 b0 ⇽ b1 ⇽ b2 ⇽ b3 ⇽ X
```

Block a3 is killed, resulting in the orphaning of b3, b4, and b5:

```
a0 ⇽ a1 ⇽ a2 ⇽ a3' ⇽ a4'
  \                     \  
 b0 ⇽ b1 ⇽ b2
```

The second case is when a parent has a conflicting idea about what the
child chain is. Here our node receives block a5, which has a conflicting
idea of what child b2 is:

```
a0 ⇽ a1 ⇽ a2 ⇽ a3' ⇽ a4' ⇽ a5
  \                     \  
 b0 ⇽ b1 ⇽ b2' -----b2'------ b2
```

As the parent always wins, even multiple blocks can get killed off this
way:

```
a0 ⇽ a1 ⇽ a2 ⇽ a3 ⇽ a4
  \                     \  
```

This behavior is easier to understand if you say instead that the node learned about block b2', which had more total work than b2 as the sum total of work done in the parent chain in blocks specifying the that particular child chain is considered before comparing the total work done in only the child chain.

It's important to remember that the parent blockchain has and validates both children's block headers; it is not possible to mine a block with an invalid secret of child headers. For instance with the following:

```
   a0 ⇽ a1 ⇽ a2 ⇽ a3 ⇽ a4
       \         \         \        \        \       
    b0 ⇽ b1 ⇽ b2 ⇽ b3 ⇽ b4 ⇽ b5 ⇽ b6 ⇽ b7
```

I can't mine block a5 that says following b2 is b2' in an attempt to kill off b2 through b7.

**Token transfer with tree-chains**

How can we make use of this? Let's start with a simple discrete token transfer system. Transactions are simply:

```c
struct Transaction:
    uint256 prevTxHash
    script prevPubKey
    script scriptSig
    uint256 scriptPubKeyHash
```

We'll say transactions go in the tree-chain according to their prevTxHash, with the depth in the tree equal to the depth of the previous output. This means that you can prove an output was created by the existence of that transaction in the block with prefix matching \( H(tx.prevTxHash) \), and you can prove the transaction output is unspent by the non-existence of a transaction in the block with prefix matching \( H(tx) \).

With our above re-organization rule everything is consistent too: if block b_i contains tx1, then the corresponding block c_j can contain a valid tx2 spending tx1 provided that c_j depends on a_p and there is a path from a_p to b_(i+k). Here's an example, starting with tx1 in c2:

```
   b0 ⇽ b1 ⇽ b2 ⇽ b3 ⇽ b4 ⇽ b5 ⇽ b6 ⇽ b7
```

```
    a0 ⇽ a1 ⇽ a2
       \        \        \        
    b0 ⇽ b1
```

Block b2 below can’t yet contain tx2 because there is no path:

\[
\begin{align*}
&b_0 \leftrightarrow b_1 \leftrightarrow b_2 \\
&a_0 \leftrightarrow a_1 \leftrightarrow a_2 \\
&c_0 \leftrightarrow c_1 \leftrightarrow c_2
\end{align*}
\]

However now c3 is found, whose PoW solution was also valid for a3:

\[
\begin{align*}
&b_0 \leftrightarrow b_1 \leftrightarrow b_2 \\
&a_0 \leftrightarrow a_1 \leftrightarrow a_2 \leftrightarrow a_3 \\
&c_0 \leftrightarrow c_1 \leftrightarrow c_2 \leftrightarrow c_3
\end{align*}
\]

Now b3 can contain tx2, as b3 will also attempt to create a4, which depends on a3:

\[
\begin{align*}
&b_0 \leftrightarrow b_1 \leftrightarrow b_2 \leftrightarrow b_3 \\
&a_0 \leftrightarrow a_1 \leftrightarrow a_2 \leftrightarrow a_3 \\
&c_0 \leftrightarrow c_1 \leftrightarrow c_2 \leftrightarrow c_3
\end{align*}
\]

Now that a3 exists, block c2 can only be killed if a3 is, which would also kill b3 and thus destroy tx2.

Proving transaction output validity in a token transfer system

How cheap is it to prove the entire history of a token is valid from genesis? Perhaps surprisingly, without any cryptographic moon-math the cost is only linear!

Remember that a transaction in a given chain has committed to the chain that it can be spent in. If Alice is to prove to Bob that the output she gave him is valid, she simply needs to prove that for every transaction in the history of the token the token was created, remained unspent, then finally was spent. Proving a token remained unspent between blocks \( b_n \) and \( b_m \) is trivially possible in linear size. Once the token is spent nothing about blocks beyond \( b_m \) is required. Even if miners do not validate transactions at all the proof size remains linear provided blocks themselves have a maximum size - at worst the proof contains some invalid transactions that can be shown to be false spends.

While certainly inconvenient, it is interesting how such a simple system appears to in theory scale to unlimited numbers of transactions and with an appropriate exchange rate move unlimited amounts of value. A possible model would be for the the tokens themselves to have power of two values, and be split and combined as required.
The lost data problem
=======================

There is however a catch: What happens when blocks get lost? Parent blocks only contain their childrens' headers, not the block contents. At some point the difficulty of producing a block will drop sufficiently for malicious or accidental data loss to be possible. With the "parent chain wins" rule it must be possible to recover from that event for mining on the child to continue.

Concretely, suppose you have tx1 in block c2, which can be spent on chain b. The contents of chain a is known to you, but the full contents of chain b are unavailable:

```
| a0 <-> a1 <-> a2 <-> a3 <-> a4 <-> a5 |
| c0 <-> c1 <-> c2 <-> c3 <-> c4 <-> c5 |
```

Blocks a3 and a4 are known to have children on b, but the contents of those children are unavailable. We can define some ratio of unknown to known blocks that must be proven for the proof to be valid. Here we show a 1:1 ratio:

```
| b0 <-> b1 (b) (b) b2 <-> b3 <-> b4 <-> b5 <-> b6 <-> b7 |
| a0 <-> a1 <-> a2 <-> a3 <-> a4 <-> a5 <-> a6 <-> a7 <-> a8 <-> a9 |
| c0 <-> c1 <-> c2 <-> c3 <-> c4 <-> c5 <-> c6 <-> c7 <-> c8 <-> c9 |
```

The proof of now shows that while a3 and a4 has b-side blocks, by the time you reach b6 those two lost blocks were in the minority. Of course a real system needs to be careful that mining blocks and then discarding them isn't a profitably way to create coins out of thin air - ratios well in excess of 1:1 are likely to be required.

Challenge-response resolution
==============================

Another idea is to say if the parent blockchain's contents are known we can insert a challenge into it specifying that a particular child block be published verbatim in the parent. Once the challenge is published further parent blocks may not reference that children on that side until either the desired block is re-republished or some timeout is reached. If the timeout is reached, mining backtracks to some previously known child specified in the challenge. In the typical case the block is known to a majority of miners, and is published, essentially allowing new miners to force the existing ones to "cough up" blocks they aren't publishing and allow the new ones to continue mining. (obviously some
care needs to be taken with regard to incentives here)

While an attractive idea, this is our first foray into moon math. Suppose such a challenge was issued in block a2, asking for the contents of b1 to be published. Meanwhile tx1 is created in block c3, and can only be spent on a b-side chain:

\[
\begin{align*}
b0 &\rightarrow b1 \\
a0 &\rightarrow a1 \leftarrow (a2) \rightarrow a3 \\
c0 &\rightarrow c1 \leftarrow c2 \rightarrow c3
\end{align*}
\]

The miners of the b-chain can violate the protocol by mining a4/b1', where b1' appears to contain valid transaction tx2:

\[
\begin{align*}
b0 &\rightarrow b1 \quad b1' \\
a0 &\rightarrow a1 \leftarrow (a2) \rightarrow a3 \rightarrow a4 \\
c0 &\rightarrow c1 \leftarrow c2 \rightarrow c3
\end{align*}
\]

A proof of tx2 as valid payment would entirely miss fact that the challenge was published and thus not know that b1' was invalid. While I'm sure the reader can come up with all kinds of complex and fragile way of proving fraud to cause chain a to be somehow re-organized, what we really want is some sub-linear proof of honest computation. Without getting into details, this is probably possible via some flavor of sub-linear moon-math proof-of-execution. But this paper is too long already to start getting snarky.

Beyond token transfer systems
================================

We can extend our simple one txin, one txout token transfer transactions with merkle (sum) trees. Here's a rough sketch of the concept:

\[
\begin{align*}
\text{input #1} &\rightarrow \text{output #1} \\
\text{input #2} &\rightarrow \text{output #2} \\
\text{input #3} &\rightarrow \text{output #3} \\
\text{input #4} &\rightarrow \text{output #4}
\end{align*}
\]

Where previously a transaction committed to a specific transaction output, we can make our transactions commit to a merkle-sum-tree of transaction outputs. To then redeem a transaction output you prove that enough prior outputs were spend to add up to the new output's value. The entire process can happen incrementally without any specific co-operation between miners on different parts of the chain, and inputs and outputs can come from any depth in the tree provided that care is taken to ensure that reorganization is not profitable.
Like the token transfer system proving a given output is valid has cost linear with history. However we can improve on that using non-interactive proof techniques. For instance in the linear token transfer example the history only needs to be proven to a point where the transaction fees are higher than the value of the output. (easiest where the work required to spend a txout of a given value is well defined) A similar approach can be easily taken with the directed-acyclic-graph of multiple-input-output transactions. Secondly non-interactive proof techniques can also be used, again out of the scope of this already long preliminary paper.

1) "Disentangling Crypto-Coin Mining: Timestamping, Proof-of-Publication, and Validation",
http://www.mail-archive.com/bitcoin-development%40lists.sourceforge.net/msg03307.html

--
'peter'[:.-1]@petertodd.org
00000000000000002fd949770524eea54446adb70603a90a4c493d345f890e04

(signature.asc
Description: Digital signature

Learn Graph Databases - Download FREE O'Reilly Book
"Graph Databases" is the definitive new guide to graph databases and their applications. Written by three acclaimed leaders in the field, this first edition is now available. Download your free book today!
http://p.sf.net/sfu/13534 NeoTech

Bitcoin-development mailing list
Bitcoin-development@lists.sourceforge.net
https://lists.sourceforge.net/lists/listinfo/bitcoin-development

Navigate to other messages

- Previous message
- View by thread
- View by date
- Next message

Navigate to other messages within this thread

- Re: [Bitcoin-development] Handling miner adoption gr...
- Re: [Bitcoin-development] Handling miner adoption gr...
- Re: [Bitcoin-development] Handling miner adoption gr...