Introduction

An attempt is made to define a strategy that will produce non-fungible time consensus.

The Problem

The problem being solved is taken from the problems section of the wiki on Etherium's github repository.

1. Timestamping

An important property that Bitcoin needs to keep is that there should be roughly one block generated every ten minutes; if a block is generated every day, the payment system becomes too slow, and if a block is generated every second there are serious centralization and network efficiency concerns that would make the consensus system essentially nonviable even assuming the absence of any attackers. To ensure this, the Bitcoin network adjusts difficulty so that if blocks are produced too quickly it becomes harder to mine a new block, and if blocks are produced too slowly it becomes easier.
However, this solution requires an important ingredient: the blockchain must be aware of time. In order to solve this problem, Bitcoin requires miners to submit a timestamp in each block, and nodes reject a block if the block's timestamp is either (i) behind the median timestamp of the previous eleven blocks, or (ii) more than 2 hours into the future, from the point of view of the node's own internal clock. This algorithm is good enough for Bitcoin, because time serves only the very limited function of regulating the block creation rate over the long term, but there are potential vulnerabilities in this approach, issues which may compound in blockchains where time plays a more important role.
Problem
: create a distributed incentive-compatible system, whether it is an overlay on top of a blockchain or its own blockchain, which maintains the current time to high accuracy.
Additional Assumptions and Requirements

- All legitimate users have clocks in a normal distribution around some "real" time with standard deviation 20 seconds.
- No two nodes are more than 20 seconds apart in terms of the amount of time it takes for a message originating from one node to reach

any other node.

- The solution is allowed to rely on an existing concept of “N nodes"; this would in practice be enforced with proof-of-stake or non-sybil tokens (see #9).
- The system should continuously provide a time which is within 120s (or less if possible) of the internal clock of >99% of honestly participating nodes. Note that this also implies that the system should be self-consistent to within about 190s.
- The system should exist without relying on any kind of proof-of-work.
- External systems may end up relying on this system; hence, it should remain secure against attackers controlling < 25% of nodes regardless of incentives.

While the problem being solved was thusly defined, the actual solution has some substantially different parameters, much of which have been generalized.

Premise and Assumptions

The problem assumes that there is network composed of N nodes, which can communicate with each other reliably. The solution relies on several assumptions, many of them stemming from the initial problem. They are:

- The network size N is sufficiently large (roughly >2000 active nodes, more on that later)
- Messages broadcast from any node will be reliably heard no more than some period of time $\Delta_{\operatorname{net}}$ after they were initially transmitted.
- This network performance does not decrease, even if every node in the network were to each broadcast a message simultaneously. Yeah, this is a fun one.
- The actual delay for a given message between two nodes is random and follows a known distribution whose shape, formula, relevant moments etc. are known.[1]
- Each node has a clock which measures the cumulative passage of time at the same rate as that of every other node.
- The population of honest clocks reporting

\Delta {\operatorname{net}} = 0

are normally distributed with a known

standard deviation of \sigma

.

- At any given block in the chain, no more than k

reputable nodes

are malevolent or inactive, where 2 k + 1 = N

.[2]

- Nodes never timestamp their own messages. Self-timestamping

invalidates a message and is an offense punished by the network.

- All meaningful messages are broadcast to all nodes in the network.

That is not to say that less distributive transmissions cause

failure, but rather that they risk being functionally disregarded by

the network.

- All meaningful messages are signed by their sendors. Unsigned or

malsigned messages are to be disregarded.

- At least one node in the network is altruistic. Objectively

speaking, if 2000 nodes can’t meet this requirement, the blockchain

probably deserves to fail.

Some futher defintions are provided below:

// TODO (active nodes, reputable nodes, nonreputable nodes, honest

nodes)

The Strategy

The strategy is composed of 5 key elements. They are:

1. Modeling Algorithm, used to interpret a sample of timestamps.
2. 3-Stage Voting, used to achieve consensus.
3. Keystone Genesis, used to bootstrap consensus of the genesis block

timestamp, providing a fixed reference point.

1. Recursive Improvement, using the existence of even low-accuracy

concurring timestamps to improve the performance of the system,

allowing for the creation of more accurate and secure timestamps.

1. Reputation System, used to incentivize and minimize the impact of

common attack strategies.

Modeling Algorithm

Diving right in:

Contaminated Sample: A statistical sample of which an unknown subset has

been falsified by intelligent attackers in a way that leaves no distinguishing characteristics between the subset and the set except for the data itself. All elements of that subset are 'contaminated values'.

Sifting Model: A statistical model attempting to obtain information from only the non-contaminated elements of a contaminated sample.

Intelligent Attacker: An actor with complete knowledge of the sifting model that will be applied to the data, who acts with the intent to alter the infromation obtained from the sifting model in some way, shape, or form. All intelligent attackers are potentially conspiratory and cooperative with each other.

It is impossible for a sifting model to obtain uncompromised information from a contaminated sample whose contaminated values make up more than half of the sample.

Theorem 4 is a natural extension of May's Theorem (which essentially states that majority rule is the only "fair" binary decision rule), applied to contaminated samples.

If there is nothing known about the population distribution of the uncontaminated elements of a contaminated sample, no effective siftng model can be created.

An effective sifting model's maximum tolerance of a sample's deviation from its population distribution and the model's maximum error from contamination are inversely proportional.

An effective sifting model's maximum contamination error is directly proportional to the variability of the uncontaminated population distribution being modeled.

Corollary 7 is key, because it makes it imperative that the variability of the clocks in the network be reduced as much as possible. This will be done in subsection 5.

The Basic Sifting Model

Despite the subsubsection title, creating even the most basic model took several weeks of contemplation, testing, and math. Omitting the somewhat circular journey and cutting to the chase, there are exactly five general steps the basic model that was implemented in R. Other practical sifting models will likely need to follow these same steps, even if they are taken in different ways.

1. Generate a weighted and unweighted kernel density function of the sample with a fixed bandwidth. The weights are derived from the

reputation of the node providing the data.[3]

1. Approximate the kernel density function with a smoothed spline f_s (x)

generated from a series of coordinates on the kernel density

function that include the range of the sample data. This is done

because actually calculating the kernel density function every time

you need it in the next few steps would be a terrible nightmare.

Generate another, f_{s_w} (x)

for the weighted kernel density

function.

1. If the population has a propability distribution

D \left(\mathhtt{} x, \mu \right)

, let the modeling function be defined as

f_m (x, \bar{x}, k) = kD (x, \bar{x})

.

1. Maximize \int^{\infty}_{- \infty} F (x, \bar{x}, k) dx

with respect to both \bar{x}

and k

, where

F (x, \bar{x}, k) = \left{ \begin{array}{ll} f_m (x, \bar{x}, k) {,}& f_s (x) \leqslant f_m (x, \bar{x}, k)\ I (f_m), & f_s (x) > f_m (x, \bar{x}, k) \end{array} \right}

where I

is a function, typically

less than 0, that varies depending on the exact implementation. Note

that 0.5 < k \leqslant 1

(as per Theorem 4), and the range of

\bar{x}

is limited by the contaminated sample.

1. Using the values of

\bar{x}

and k

from step 4, derive the

score function as follows:

{\operatorname{score}} (x_{\operatorname{reputable}}) = \left{ \begin{array}{ll} 1 {,}& f_m (x, \bar{x}, k) \geqslant f_s (x)\ \frac{f_m (x, \bar{x}, k)}{f_s (x)} & f_m (x, \bar{x}, k) < f_x (x) \end{array} \right}

The score function for disreputable nodes is similar, but uses a weighted balance between {\operatorname{score}} (x_{\operatorname{reputable}})

and a version of {\operatorname{score}} (x_{\operatorname{reputable}})

that uses f_{s_w}

instead of f_s

.

The score functions are later used to augment the reputability of a node based on its reports.

An Implementation Thereof

// todo, put more here, but basically:

- While my educated guess for an optimum bandwith for a normally distributed sample is $\frac{\sigma}{e}$, by no means was a rigorous study of the accuracy of the final results with various bandwiths performed.
- The Implementation function I that seemed to work best, based on my tests involving generalized contaminated populations (I'll provide the R code I used at some point, when I'm less trying to get to bed ASAP, but basically I plugged the function [normal model*proportion of good nodes + some attacking normal model (with a different μ and σ) * proportion of attacking nodes] in for f_s in the above steps), was I (f_m) = - $\frac{ef_m + 1}{e + 1}$
- While my results were satisfactory enough for me to write up the report and publish it, my tests included numerous instances where terrible values for $\bar{x}$ and k were chosen. However, I'm convinced these were due to limitations in the heuristic optimization algorithm in R's stats library rather than failings of the sifting model per se.

The Voting Process

The voting process, or alternatively voting cycle, must be broken down into three distinct steps.

1. Vote, where nodes announce the times at which they observed all meaningful messages they received over the previous block of time.
1. Sharing of Registries, where nodes announce the list of votes they think are valid and were cast within the acceptable voting period.

1. Chorus of Consensus, where the network finalizes the block to be added to the chain with a majority of the reputable blocks reaching the exact same conclusion independently.

The initation step, not listed, is just that the predefined amount of time has passed since the second to last voting cycle. The timestamp for a given voting cycle is defined as the mean of the timestamps of the votes from that cycle that were included as valid votes.

The summary step, also not listed, is simply one or more altruistic nodes revealing the solution that consensus was reached upon. This is done for posterity, simplicity, and validation.

The Vote

//todo fill this out when I'm not rushing to bed

The Sharing of Registries

Wow this is going to be kinda hard to explain without explaining the vote. The votes all happen in a window of time, but everybody's perception of that window is relative to their clock. So when the last votes are coming in, some nodes might think that they're on time but others might not. Pretty simple concept. While the majority of this problem is solved with intelligent error acceptance, creating the window that I just suddenly started talking about, there's a slight hiccup.

If even just ONE node sends a poorly/aptly (depending on your perspective) timed vote (as in one just shortly after the window closes, from the perspective of the nodes who think it closes first), then there will be a massive disagreement on whether or not to include the vote in the contaminated sample that is plugged into the sifting model defined in section 2.1. As a result, they won't be able to independently reach the same conclusion, ruining the consensus.

That's where the sharing of registries comes in. Each node shares the list of votes that it thinks happened in the window. This registry sharing process has its own window, after which nodes assume that any nodes that haven't shared their registry aren't going to. Votes that are included on the majority of the registries are included in the final vote.

Registries after the window are still to be accepted. More explanation to come.

Oh, also, only reputable nodes' registries are counted, and the expectation is that the nodes that vote are the same nodes who submit

registries.

The Chorus of Consensus

Basically, everyone solves everything, validates stuff, calculates the timestamps, and figures out the block. Nobody is rewarded for completing the block, because everyone is expected to derive the block independently. Instead, everyone who correctly completes the block independently and announces the correct answer gets rewarded. As answers pour in, nodes can try to match the answers by keeping & excluding registries from the latter half of the registry list. The only discrepencies in the registry listings should be from any late/early registry senders (who of course can be punished).

This is solved by having nodes submit answers without showing their work, and allowing them to submit multiple answers (within reasonable limitations, of course). Thus, they may change their answers. If multiple answers are submitted, the earliest (determined by it's position in the node's local chain (every message a node sends contains a reference to the previous message the node sent)) answers that reach consensus are used.

Nodes release answers without showing their work by hashing the block with their public key, and signing that. If your block answer + their public key = their hash, you got the same answer wooo.

Earliest majority answer wins. The timestamps of the answers shouldn't matter.

Keystone Genesis

The intial block is kinda tricky. There's a few ways that it is different from a typical block. Everyone expects one, single, trusted public key to be the "genetor". The genetor's sole purpose is to validate that this blockchain is in fact the correct blockchain. Nodes in the inital network are assumed to be reputable.

The genetor starts the system by releasing a special signed 'start' message.

Instead of waiting to vote, every node immediately acknowledges the receipt of this start message, referencing the start message in their ack vote as proof it was sent after kickstart. This provides a lower limit on the times, allowing for the nodes to safely perform a voting cycle. The only other requirement is that the genetor is one of the submittors of the final answer, so that the start message cannot be used

to create other nodes

Oh yeah, I forgot to mention. Blocks contain: transactions w/ timestamps, timestamps, rewards & updates for the previous voting cycle, and all the voting data used to derive the timestamps.

So yeah, by providing the same answer, the genetor proves that they were present for that particular set of initial votes. Past that, everything should be the same. The nodes treat the timestamp of the start message as the second-to-last vote mean thing when determining when to vote next.

Recursive Improvement

Okay, this is where things go from cool to super cool, in my humble opinion.

Because nodes in the network have reached an agreed upon, shared time, and have a record of what they thought the time was vs. what the network thought it was, they could adjust their clocks to better match the consensus. The error in the adjusted clock is limited by the attacking error combined with the network error. This should result in a much tighter distribution. While the security benefits of this cannot be easily incorporated into a given blockchain implementation (because presurring nodes to set their clocks to the network clock ruins the distribution of clocks, so that distribution has to be modelled too and the transition from one model to the other (and one clock to the other) has to be predefined and followed by all nodes simultaneously), one clock blockchain could piggyback on the clock of another to improve their accuracy at the start, and then continue on with that improved accuracy indefinitely.

However, even without this, a blockchain can still garner several of the benefits of the modified clocks by demanding nodes keep track of two clocks. The first is the unaltered clock they measured with at start, and the other is their network adjusted clock. Scheduling semantics, such as when to vote and register, can be reasonably expected to occur within the accuracy range of the network adjusted clock. Meanwhile, data values will be expected to be reported with the accuracy of the unaltered clock.

Reputation

blah blah blah,

something I haven’t really mentioned I don’t think is that any implementation that stays with a bifurcated clock schema must also watch nodes carefully for changes. An honest node’s differences from the network clock will be fairly consistent, to an error of approximately (\Delta \ensuremath{\operatorname{net}}). It’s important to hold nodes to that honesty to prevent Sybil attacks that play for the long game. Similarly, to prevent long-range attacks, strong activity requirements must be included in the reputability scheme, so that idle nodes are removed quietly. Quick warm up and quick cooldown should be best, methinks. It’s just very very important that no attacker ever in history has >50 % of the reputable nodes, because then they can perform long-range attacks (kinda, they’d still not be able to fool the active nodes, but they would be able to confuse the heck out of an outsider). Further protection of that could be some other scheme, perhaps proof of work or proof of stake, but it shouldn’t be necessary. This is because rewards are dependent on reputation. The most likely thing I can see it needing is a benchmarked (via the network clock) bonus proof of work option. Nodes that opt into it can gain more rewards & raise their reputation cap by doing a little extra work with each message. This strategy would be similar to the e-mail one the proof of work puzzle came from. Also, if this is done, I highly highly highly advise a chain of low-difficulty puzzles rather than one high-difficulty puzzles. More low-difficulty puzzles together makes a tighter distribution of puzzle completion time. Honestly, I’m a little surprised more proof of work blockchains don’t already do this. The principle is the same, it just causes the puzzle duration to be far more consistent. It also allows for far more control over the expected duration: the expected duration can be extended or shortened by just adding or subtracting one more small puzzle, rather than by doubling or halving the puzzle’s difficulty.

One major thing I forgot to mention is that I’m pretty sure this scheme can also remove the need for a currency to incentivize the blockchain, so long as reputability itself has value. For instance, if reputability was linked to say, access to a game network, or access to really any service that is supported by the transactions tracked on the network. Another example would be a social network where posts can only be made and retrieved by reputable nodes.

1. The algorithm test/demonstration was performed under the quite moronic assumption that the network delays are completely random and normally distributed. It’d be most accurate to say that consideration of the network delay was practically abandoned in its entirety. That said, implementing a similar algorithm to the one detailed that is based on a non-normal distribution

shouldn’t be too difficult and is left as an exercise for the reader.

1. That said, I think the system would sleep a lot better at night if the attacking population was < 33%. As you get past that, it just gets too easy to significantly influence the outcome of the model. Of course, this is a subjective cutoff on a smooth continuum, but
1. While the weighted version & information derived from it technically are outside the definition of a sifting model, it seems unlikely that any realistic implementation would avoid their usage.