Taraxa White Paper

Building IoT's Trust Anchor

Version 0.9.2

May 1, 2019

The Taraxa Team

contact@taraxa.io

Disclaimer: this white paper is intended as a preliminary technical overview of the Taraxa protocol and ecosystem and is not meant to be comprehensive or fully finalized.

Abstract

The advancement of IoT ecosystems has been consistently held back by technical challenges in security, maintenance, data provenance, and incompatible standards, as well as by business challenges in a generally lacking or lackluster use cases, strategically-risky platform lock-ins, and intentionally-siloed data management, and an increasing social awareness and demand for better privacy protection as devices become ubiquitous. Blockchain technology can help to address many of these challenges by building trust anchors into the ecosystem, thereby granting devices operational and economic independence, an awareness for asset ownership, and the capability to freely trade with one another.

In this whitepaper, we introduce Taraxa, a public ledger focused on building trust anchors for IoT ecosystems with the following innovations,

- **Rapidly-finalizing DAG** to maximize throughput and minimize inclusion & finalization latency
- **Fuzzy sharding** to minimize wasted work and maximize network-wide parallel processing
- **Speculative concurrency** to minimize transaction execution latency
- **Adaptive protocol** to help the network learn and adjust its governing parameters on the fly
- **Trustless light nodes** that can verify what it's been told by full nodes

Table of Contents

1. Introduction....4
   1.1 The Promise of IoT....4
   1.2 How Blockchain Technology Could Help....5
2. Review of Existing Projects....6
3. Taraxa Architecture....6
   3.1 Design Principles....7
   3.2 Ledger Architecture Overview....7
   3.3 Concurrent VM Architecture Overview....8
4. Core Consensus....9
   4.1 Brief Background....9
   4.2 Ordering via Anchor Chain....10
   4.3 Rapid Finalization....13
   4.4 Fuzzy Sharding....14
   4.5 Adaptive Protocol....16
5. Concurrent Smart Contracts....16
   5.1 Brief Background....16
   5.2 Speculative Execution....17
   5.3 Concurrent Schedules....18
   5.4 Conflict Detector....19
6. Technical Roadmap....20
7. Economics....20
8. References....21

1. Introduction

1.1 The Promise of IoT

Long has the prospect of billions upon billions of intelligent, connected, and autonomous devices freely communicating and coordinating with one another been held up as an unquestioned vision for the future. The confluence and rapid advancement of technologies such as ever-more precise sensors and actuators, cloud computing, and near-ubiquitous connectivity have all made this vision seem increasingly likely. The market is full of audacious predictions of the world full of trillions of connected devices and the market for IoT reaching hundreds of trillions of dollars in market size.

Yet, with each passing year, that promise has remained just that, a promise.

For IoT to fully realize its promised potential, we would require a far more rapid pace in collecting and sharing of that collected data. IoT's core value lies in the insights generated from the data collected by devices, which then in turn enable devices and humans to act upon these insights to generate value, eventually driving towards completely autonomous behaviors. As it stands today, the rate at which new data is being collected is slow, and the rate of sharing (and trading) of data is glacial. While by and large the sector has moved beyond the basic "what is IoT" understanding gap, it is nevertheless being held back by a set of very practical obstacles.

Technical obstacles

Existing IoT infrastructures face a myriad of challenges as they scale up and eventually reaches trillions of autonomous devices. Centralized governance systems are ill-equipped to handle security as they present a single point of failure, they tend to scale poorly with workload as most device to device interactions tend to be localized, data collected by devices face questions of authenticity and provenance, and manufacturers in an attempt to create moats around their markets intentionally build in incompatible communication standards resulting in huge numbers of fragmented silos that take herculean costs and efforts to integrate.

Business obstacles

Due to the fundamental nascent nature of IoT systems as well as the various technical challenges, business cases for deploying large-scale IoT networks are often difficult to justify. The effort to discover financially-sound business cases are often hampered by data sensitivities tied to a fear of leaking trade know-how. Finally, with the rise of mega platforms and infrastructure players, businesses are becoming increasingly aware of the strategic risk of tying themselves into these centralized gardened walls, leaving them with a significantly-weakened bargaining position.

Social obstacles

With the rapid proliferation of digitized technologies, the public at large has become increasingly aware of the omnipresence of data-collecting sensors as well as concerned about how they're being used. Recent data leaking scandals [1] [2] and the EU's GDPR [3] have all placed privacy and data ownership at the center of civil discourse. If IoT as a technology is to continue proliferation, it must address data privacy concerns head-on and provide socially-acceptable solutions to guarantee secure data ownership and usage without triggering innovation-killing regulatory backlashes.

These and many more obstacles prevent the rapid adoption and value actualization of IoT in general.

1.2 How Blockchain Technology Could Help

Blockchain technology (for this section the term blockchain is used to refer generally to decentralized ledger technologies) has given us a tool for a decentralized set of participants to collaborate, trade, and compete without preestablished trust, radically diminishing or in many cases completely removing the need for centralized systems to provide guarantees and coordination. This decentralization has several interesting consequences as applicable to IoT.

Fractionalizing Resources

Decentralization enables society at large to unlock underutilized assets such as storage, bandwidth, and computing power. Think of it like a shared economy without the centralized platform (shared riding without Uber, shared rooms without Airbnb), where idle resources could be more fully leveraged. This type of sharing also has the added effect of lowering the minimum purchase size for the individual or the small business owners, for whom in the past setting up an IoT infrastructure is prohibitively expensive.

Decentralized Security and Responsibility

No longer are centralized coordination authorities necessary to facilitate transactions or maintenance updates. Blockchain enables fully decentralized networks (without single points of failure) that is far more secure and transparent. Hacking a single device wouldn't get you access to a million IoT devices, just one, and the rest of the network hums along, significantly raising overall network security. Each device and device owner are now masters of their own device, its security, and the data it produces, making network maintenance a far more manageable task.

Open Source Ecosystem

One of the core features of public ledger ecosystems is that the underlying technology is all open-source, without which participants cannot be sure to trust the code that runs the ecosystem. This type of open-source ecosystem not only minimizes the risk of becoming platform-dependent (anyone can replicate the ecosystem at any given time if they choose to do so). For IoT data owners who may be reluctant to rely upon external platforms for mission-critical analytics, they can be sure that platform is fully decentralized, transparent, and free for anyone to duplicate, should they find any part of the ecosystem to their dissatisfaction.

Democratization of Data

Blockchain can eventually enable devices to trade data with one another, enabling the much-heralded machine-to-machine economy. In time this economy will develop its own assets, pricing and reputations, each device its own business. With more sophisticated AI even generate its own algorithms to make use of the data available to maximize its own interests. By democratizing data, innovation (and business cases) would more readily and spontaneously blossom as ever more participants are granted access.

Integrating blockchain technology into IoT would prove a significant boon to the space, enabling more data to be more freely shared and traded, triggering an explosive emergence of hitherto unthought-of applications.

2. Review of Existing Projects

There are several blockchain projects purported to target the IoT space. Here we take a brief look at a few well-known projects.

- **IOTA** uses a DAG-based Tangle instead of a linear ledger. It is miner-less and fee-less, achieved by having every node perform the tasks of verifying each other's transactions. However, a finalized consensus protocol has yet to be published, currently relying on a closed-source, centralized coordinator to order and validate transactions.

- **HDAC** is based on MultiChain, with a focus on mediating transactions and permissions between various public and private blockchains, with HDAC adding innovations such as an ePOW consensus in its private chain implementations.

- **VeChain** is a DPoS-based chain, with permission-less participation but permissioned miner eligibility and governance model. Its applications are primarily focused on product lifecycle management through IoT data anchoring.

- **IoTeX**, adopts a chain of chains topology, enabling different IoT ecosystems to create their own sub-chains that can transact with one another through the root-chain, with a consensus algorithm similar to those proposed in Algorand and Dfinity (random selection of block proposal and validation committees). IoTeX also has functionalities to maintain transactional anonymity.

Many important IoT use cases such as data anchoring are stateless transactions, whereby the key figure of merit for the blockchain network is simply block inclusion (see Section 4) instead of full finalization. Taraxa's unique topology allows for a decoupling of inclusion as an asynchronous and independent network activity versus finalization & execution, allowing devices to get much faster feedback without having to wait for the remaining parts of the consensus.

Device transactions also heavily depend on smart contracts. Taraxa combined the benefits of a fast-advancing DAG topology and the instant and fair finality of a VRF-driven PBFT process (see Section 4), giving smart contracts with state-dependent logic a definitive guarantee of immutability – not a probabilistic one.

To support the role of smart contracts in device transactions, Taraxa's unique use of speculative concurrency (see Section 5) in the construction of a concurrent VM drastically boosts execution speed, saving precious time for the entire network of full validating nodes.

All networks evolve over time, and the best networks smoothly adapts to these evolutionary changes. Much of blockchain network's evolution instead of happening automatically have taken place in online forums and offline meetings, often regressing into vicious and nonproductive disputes. Taraxa's protocol is designed to seamlessly reach consensus on key network behavioral attributes such as block generation rate, block size, and shard jurisdictions, without all the fuss and drama.

3. Taraxa Architecture

3.1 Design Principles

Taraxa abides by the following set of design principles.

- **Maximum concurrency:** Taraxa maximizes both horizontal as well as vertical concurrency to be inclusive of all work done and make sure all hardware resources are fully leveraged in transaction processing & validation.
- **Minimum waste:** every node only does the work that's necessary, when they're randomly designated to do so, with all work done included, minimizing overlaps and waste, and all done with almost no coordination overhead.
- **Device-conscious:** Taraxa assumes that most of the nodes on the chain will be IoT gateways, with our protocol built to reflect the processing, memory, and bandwidth limitations of these devices. We will also provide hardware reference designs and work with our partners to integrate them into their systems.
- **Developer-friendly:** Taraxa aims for maximum backwards-compatibility by adopting an EVM-compatible toolchain to ensure that most developers do not need to learn a new set of languages & tools. Over time, Taraxa will adopt the WebAssembly compilation target to provide a more provably secure execution environment.

3.2 Ledger Architecture Overview

The Taraxa ledger roughly takes the following architectural approach. More details are found in Section 4 and Section 5.

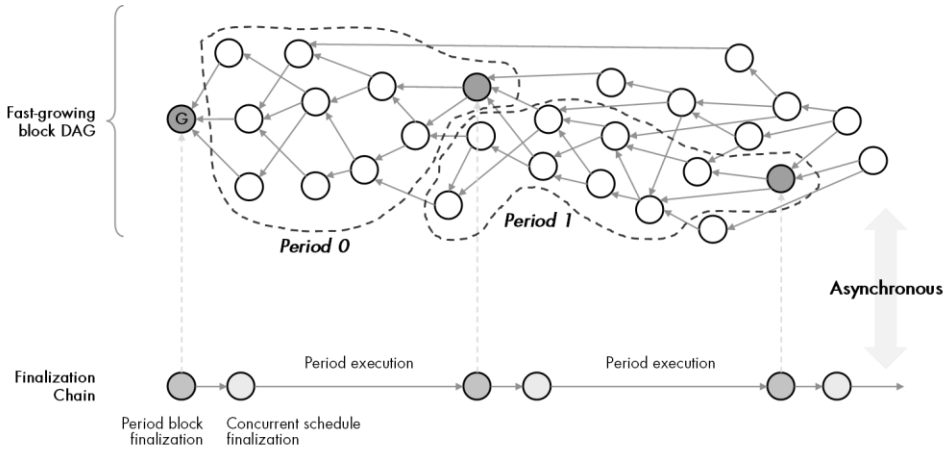

Figure 1: Taraxa Ledger Architecture

Taraxa's core architecture is divided into two parts, a block directed acyclic graph (DAG) at the top and a Finalization Chain at the bottom.

The block DAG is where blocks are proposed, validated, but not executed. This allows the DAG to grow quickly and decouples block inclusion from finalization and execution, a unique advantage to many stateless IoT use cases.

The DAG is divided into Periods, bounded by Period Blocks that are finalized through a VRF-enabled PBFT voting process. This voting process is governed by sparse blocks on a separate Finalization Chain. Once a Period Block has been finalized, all blocks belonging between two finalized Period Blocks have deterministic ordering as defined by the GHOST [4] rule.

3.3 Concurrent VM Architecture Overview

Taraxa's concurrent VM consists of the following components.

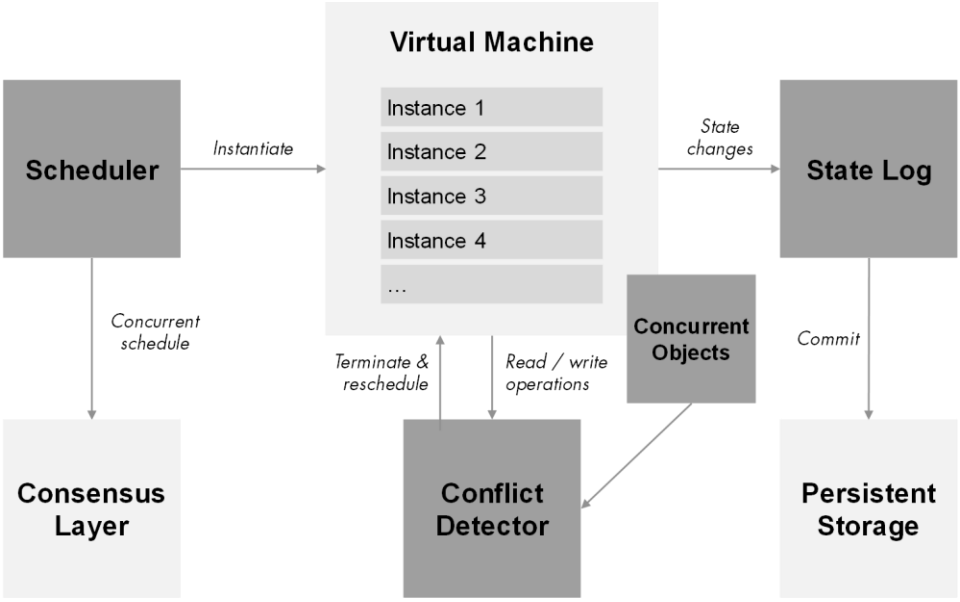

Figure 2: Taraxa Concurrent VM Architecture

There are two phases: schedule discovery and validation.

During discovery, the Scheduler takes input in the form of a block, instantiates multiple instances of the VM. The Conflict Detector monitors all read / write operations occurring while the instances are running, keeping a log of state changes and terminating instances that conflict, scheduling them for sequential execution, resulting in a Concurrent Schedule consisting of a concurrent set followed by a sequential set. In some instances, the conflicts are reported not directly by the VM instances but by a set of Concurrent Objects designed to be conflict-aware. The Concurrent Schedules are sent to the consensus layer, so the network reaches agreement.

During validation, the remaining nodes (most nodes that were not part of the consensus process) now follow the agreed-upon Concurrent Schedules to rapidly execute the blocks, resulting in significant time savings for the entire network overall.

4. Core Consensus

4.1 Brief Background

As the blockchain space matures, pioneering networks such as Bitcoin [5] and Ethereum [6] have run into scalability bottlenecks. One of the key scalability metrics is total network throughput, usually measured by transactions per second (TPS). For single-chain topologies, however, increasing TPS necessarily means a decrease in security, the recovery of which negates any TPS gains.

To increase TPS, the network could increase block size β and block generation rate γ. Increasing β necessarily increases network delay δ, which in turn reduces the likelihood of nodes all hearing the same information in a timely manner, increasing the likelihood of branching. Increasing γ has the effect of increasing the number of blocks proposed on the network, but since on a single-chain topology only a single block can ever be accepted, more blocks actually increases the options nodes have to make the incorrect bet on the longest chain, also increasing the likelihood of branching. Hence, we see a hard tradeoff between TPS and security [4].

$\beta\gamma \propto TPS$	block size and block generation increase TPS
$\beta \propto \delta$	block size increase network delay
$\delta\gamma \propto branching$	network delay and block generation increase branching (decreases security)

One elegantly simple approach is to abandon the single-chain approach and adopt an inclusive approach in the form of a DAG [7], or specifically a block DAG. In a block DAG, blocks could be proposed by multiple nodes and they would all be accepted if they were valid. Unlike in a single-chain topology, each block would reference not just a single parent, but multiple parents – in fact as many parents as the proposing node sees as tips of the current DAG. In such a DAG, there isn't a hard tradeoff between TPS and security, as the network could reliably increase β without sacrificing security. The block DAG increases β by being naturally inclusive of all branches so total throughput is naturally increased as now nodes can process transactions in parallel, security is not sacrificed through a mechanism of implicit voting (each block points to multiple previous blocks that form the tips of the DAG at the time of block proposal) coupled with the GHOST rule, combined which allow the uncoordinated honest majority of nodes on the network to defeat a coordinated malicious minority [7].

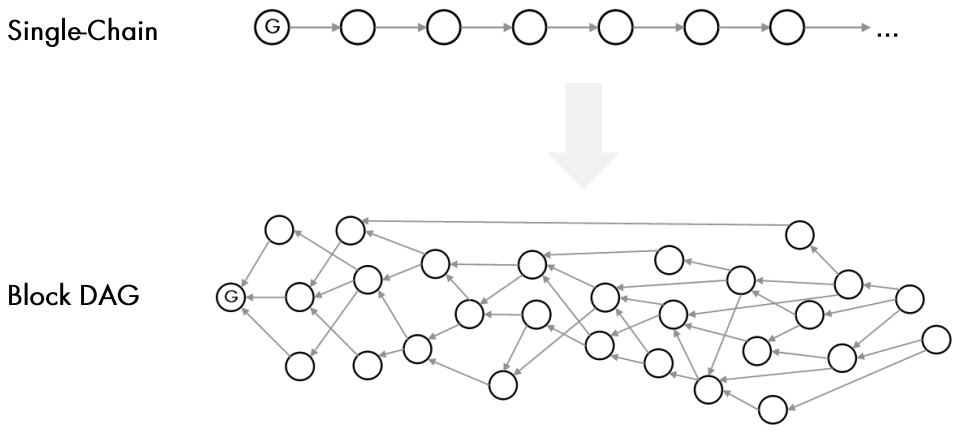

Figure 3: From single-chain to block DAG

However, the block DAG topology isn't without its own set of challenges, here are a few,

- Ordering convergence
- Finality (confirmation latency)
- Block efficiency

Taraxa sets out to address these issues.

4.2 Ordering via Anchor Chain

To ensure that the block DAG reaches rapid convergence amongst nodes, we note that not all parent block references need to be equal. The ordering algorithm originally proposed by GHOSTDAG [7] takes a two-step process: separating the DAG into blue vs. red clusters, and then using the GHOST rule to map out an induced chain within the blue cluster.

Here we note that you could remove the need for the initial clustering by simply allowing implicit voting via the block pointers on what each of the proposers think happens to be the heaviest block (analogous with B_{max} in GHOSTDAG) at the time of the proposal. This idea was first proposed by the OByte (formerly Byteball) project [8] whereby one of the parents referenced is called the "best parent". Unlike in OByte, the best parent is not determined by a predefined set of witnesses, but rather by the tool we already have at our disposal, the GHOST rule.

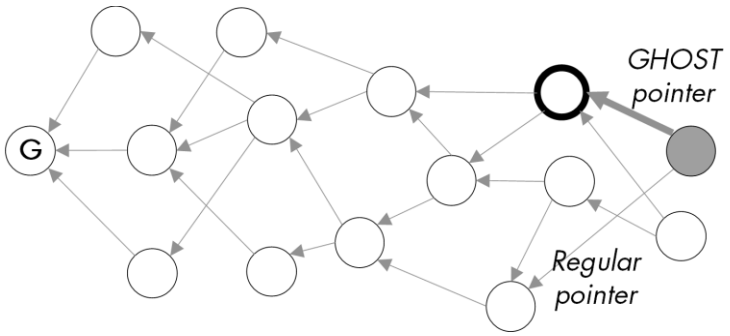

Figure 4: Block referencing the heaviest tip calculated via GHOST

At any given moment, a node can independently calculate which of the tips on the DAG it observes is the heaviest tip according to a modified version of the GHOST rule. In the original GHOST rule, all pointers are of equal weight. In Taraxa, the weight calculations only involve GHOST pointers – those that the proposing nodes calculated to be the heaviest. The remaining regular pointers are involved in total ordering. This mechanism guarantees an exponential falloff of reordering risk within the block DAG over time, and that different nodes' view of which blocks are considered the heaviest blocks rapidly converge over time.

Taraxa's use of Periods (see Section 4.3) help to bound the complexity of the weights calculations, as the traversal stops upon hitting blocks that are members of a previously finalized Period.

From any block on the block DAG by following the heaviest blocks forward (towards the newest blocks), we can construct an Anchor Chain inside the DAG, much like the Main Chain proposed by OByte.

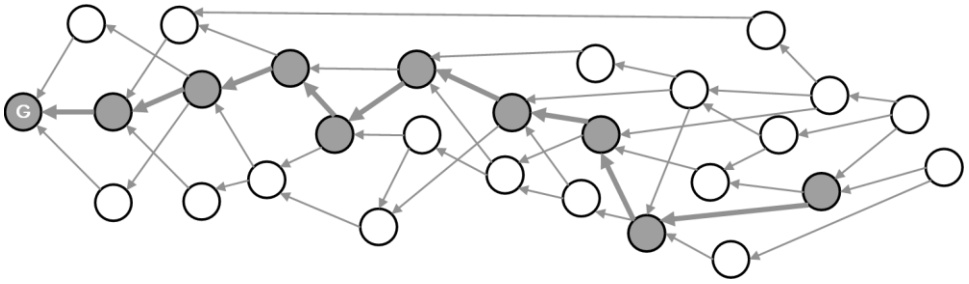

Figure 5: Anchor Chain constructed inside the block DAG

Here's a breadth-first algorithm describing how weights for a yet-to-be-finalized Period, which is executed whenever a new block is being proposed.

Algorithm 1: calculate the weights of each block in a non-finalized Period

---

Input: *S* – a set of tips visible to the node

Output: *W* – a dictionary of blocks and weights for the current non-finalized Period

```
1:  function CALCULATEWEIGHTS (S):
2:      current_layer ← S
3:      while current_layer is not empty:
4:          for each block in current_layer:
5:              parent ← block's heaviest parent block
6:              if parent does not belong to a finalized Period then
7:                  if parent is not in parent_layer then
8:                      add parent into parent_layer
9:                      insert parent into W with a weight of 1
10:                 else
11:                     increment W(parent)'s weight by 1
12:         current_layer ← parent_layer
13:         parent_layer ← {empty set}
14:     return W
15: end function
```

After the weights are calculated for the currently non-finalized Period, the Anchor Chain is constructed from the block DAG. The Anchor Chain is used to later determine total ordering between two Period Blocks, but here it is used to determine the heaviest tip on the block DAG, which is where the Anchor Chain terminates.

Below is an algorithm which determines the Anchor Chain. Here we assume that, unlike a typical DAG, there exist forward pointers from parent to the child block that has a GHOST pointer pointing back to it. In actual implementation such relationships are generated and discarded at runtime.

Algorithm 2: calculate the Anchor Chain

Input: *P* – previous Period Block, *W* – a map of blocks and weights for the current non-finalized Period

Output: *anchor_chain* – a set (that preserves insertion order) of blocks that form the Anchor Chain

```
1:  function FINDANCHORCHAIN (P, W):
2:      current_block ← P
3:      while current_block has children:
4:          children_weights ← map (block, weights) from current_block's children ∩ W on block references
5:          heaviest_child ← the block with the maximum weight in children_weights
6:          add heaviest_child to the end of anchor_chain
7:          current_block ← heaviest_child
8:      end while
9:      return anchor_chain
10: end function
```

With the Anchor Chain calculated, the heaviest tip in the block DAG is simply the final element of the Anchor Chain, which the newly proposed block will reference with the GHOST pointer.

Total ordering becomes completely deterministic once the Anchor Chain is calculated. Traverse the block DAG starting from the previously-finalized Period Block down the Anchor Chain, and for every Anchor Block, find its parents (excluding the previous Anchor Block) which constitutes an epoch. Topologically sort the epoch with tie breaking via the lowest block hash (e.g., tie breaking for blocks 3 and 4 in Figure 6) and keep moving down the Anchor Chain until it has been exhausted.

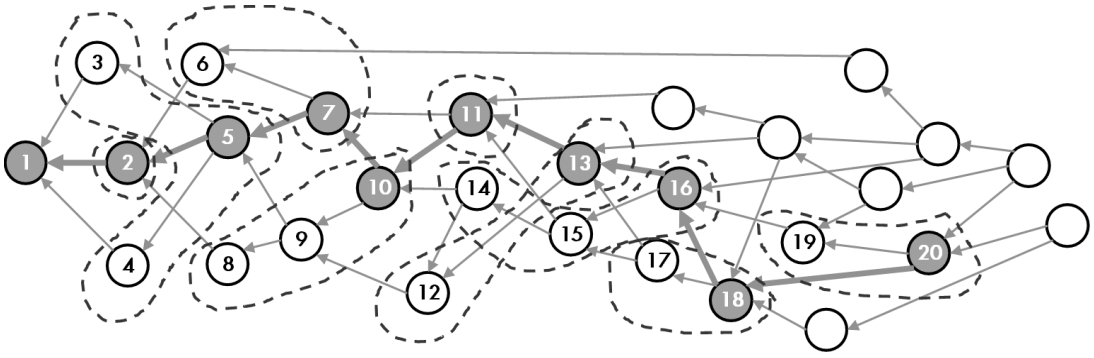

Figure 6: Total ordering of block DAG along the Anchor Chain

Algorithm 3: calculate the total ordering of the block DAG

Input: *anchor_chain* – a set of blocks that form the Anchor Chain for the currently non-finalized Period

Output: *ordering* – total ordering of the currently non-finalized Period

```
1:  function ORDERPERIOD (anchor_chain):
2:      for each anchor_block in anchor_chain:
3:          epoch ← set of parents for anchor_block, excluding the previous block in the anchor_chain
4:          sorted_epoch ← topologically-sorted epoch, with tie-breakers via lowest block hash
5:          add sorted_epoch to the end of ordering
6:      return ordering
```

7: **end function**

4.3 Rapid Finalization

In many blockchain networks, finality is a matter of probability. For example, in Bitcoin the convention is to wait for a transaction to be 6 blocks deep [9] (which takes on average 60 minutes) into the chain before accepting it as "finalized", however the risk of network reordering is never zero, and the exact probabilities depend on how much hash power an assumed attacker is.

This is also true for the block DAG, whereby the reordering risk falls off exponentially but is never truly zero. This may be acceptable for small, coin-only transactions, but often unacceptable for high-valued transactions and especially for smart contracts.

Smart contracts (more on Section 5) are logic built on top of the blockchain. Unlike simple coin transfers exhibit only a single behavior – modifying the states of two predefined accounts, a smart contract could (and often do) impact many accounts at once, many of those could themselves be smart contracts and trigger a large-scale cascade of impact. On top of which, many such smart contract implement mechanisms with branching conditions based on previous states – e.g., auctions, trading algorithms. All of this makes having a truly *finalized* state critically important.

To reach fast finalization, we use a VRF-enabled fast PBFT process first proposed by the Algorand [10] project, in which a randomized subset of the network is chosen to cast a vote. Unlike in Algorand and other similar protocols, this vote in Taraxa is far simpler and is asynchronous with block generation.

As the block DAG grows, the network takes successive votes to place infinite weight on a specific Anchor Block, which then becomes a Period Block. The voting result is encoded into a block on the Finalization Chain (see Figure 1). This vote is very simple because it is not a vote on the contents or the validity of the block – that has been achieved already when constructing the block DAG by implicit voting through gossiping the block throughout the network – but purely on whether or not this is the Anchor Block that should become a Period Block. A much simpler vote means the voting process is much faster, as there are less assertions to validate among the randomly-selected committee members. Once a Period Block has been finalized, all blocks connected between the new Period Block and the last Period Block now have a deterministically-defined (in other words, finalized) ordering, forming a new Period within the block DAG.

This finalization process is asynchronous to block generation. As the block DAG at the top grows, it is only concerned with transaction inclusion – in fact nodes do not even execute the transactions within the blocks. The block DAG is just for block validation and transaction inclusion and grows completely independently of the finalization and execution process that happens through voting.

This decoupling of transaction inclusion vs. finalization & execution has a particularly interesting use case for stateless transactions, often found in applications involving IoT sensor data anchoring. A stateless transaction is one where the transaction has no logical relationship with other transactions, hence no subsequent transactions would depend on such stateless transactions. IoT data anchoring, where an IoT device periodically hash data sets collected over time and commit them into the blockchain, is a stateless transaction. Hence a stateless transaction is only concerned with transaction inclusion, which guarantees that it has made it into the blockchain, and since any amount of subsequent reordering has no impact on their validity, an IoT device has no need to wait for a finalization signal before anchoring another set of data.

Lastly, having finalized periods across the block DAG effectively caps the computational complexity of calculating weights via GHOST, as any node only needs to recursively calculate weights until it hits a confirmed Period Block.

4.4 Fuzzy Sharding

When more than one node could successfully propose blocks as in block DAG, you could run into problems of block efficiency.

First, there could simply be too many blocks if there were no rate-limiting mechanisms in place. Classic blockchain projects like Bitcoin and Ethereum relies on Proof of Work (PoW) as a way to rate-limit block generation, but Taraxa uses a Proof of Stake (PoS) and we believe that the sheer amount of energy expended by PoW is not sustainable or socially responsible – we'd need a non-energy destroying method of limiting block generation rates.

Second, transactions contained within different blocks could overlap with one another, causing redundancy and waste. The most basic strategy to control this is to require that when a block is proposed, it contains none of the transactions included in the tips (parents) it is referencing. The proposer further has no financial incentive to reference older transactions in non-tip blocks as its block would likely either be rejected as malicious, or that these redundant transactions will be pruned during execution and proposer would have received nothing for its efforts. But this basic approach is often not enough if transactions begin to flood the network.

Taraxa implements an algorithm called Fuzzy Sharding which uses cryptographic sortition to efficiently limit block generation as well as define transactional jurisdiction to minimize transaction overlaps.

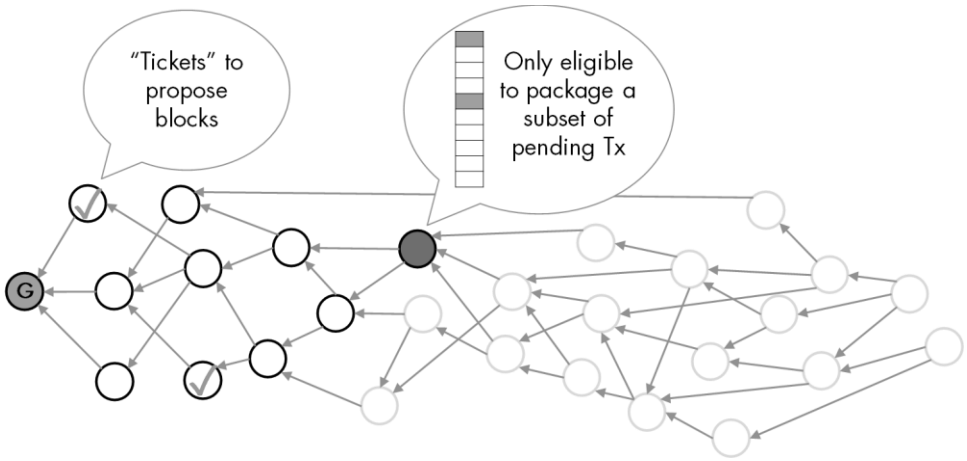

Figure 7: Block generation limitation and transaction jurisdiction definition through Fuzzy Sharding

To limit block generation, Fuzzy Sharding allows each proposer to independently calculate how many blocks they are eligible to generate. For each new block added to the block DAG, each proposer signs the block hash and then hashes the resulting signature, creating a ticket. This ticket is only valid if it falls below a certain threshold, which is defined by a network parameter and increased (increases the probability of eligibility) by the proposer's stake (or delegated stake). To mitigate a malicious proposer saving up tickets and then flooding an entire Period with its blocks, these tickets have an expiration of two (2) Periods, in that a ticket generated in P_0 is valid for P_0 and P_1, but not beyond that. They are valid for two Periods just to make sure that at the boundary between Periods, valid tickets are not invalidated due to latency issues on hearing the next confirmed Period Block. These tickets are "virtual", in that they are not included in any blocks as they could be easily validated by nodes other than the proposer by performing the exact same operation to ensure eligibility of the proposer and, by extension, the validity of the block.

Here's the simple ticket calculation algorithm and is easily validated by another node observer.

Algorithm 4: calculate the tickets available for the current Period

Input: T – set of blocks from the previous finalized and the current non-finalized Period, *threshold* – threshold under which the ticket is a winner, *stake* – proposer's coin stake, β – threshold modifier

Output: *winners* – dictionary list of winning tickets

```
1:  function FINDWINNINGTICKETS (T, threshold, stake, β):
2:      for each block in T:
3:          ticket ← hash of the node's signature of the block's ID
4:          if ticket < threshold · stake · β then
5:              add (block, ticket) to winners
6:      return winners
7:  end function
```

Note that in Algorithm 4, the threshold modifier β should be positively-correlated with the maximum hash and negatively-correlated with the total number of coins.

To define transaction jurisdiction, a proposer follows an algorithm that places it into a specific range of transaction addresses (or accounts) for which it has jurisdiction over. In other words, the proposer is only eligible to pack transactions from addresses within its jurisdiction into a new block. A proposer first signs an Anchor Block and then hashes the signature, receiving a certificate. This certificate is then mapped into the pool of pending transactions to see which ones the current node is eligible for. This could be done via a simple modulus operation, for example, as described in the simple algorithm below.

Algorithm 5: find transactions within the proposer node's jurisdiction

Input: P – pool of pending transactions, N – number of jurisdictional shards, *past_period_block_id* – the id of a past period block (e.g., the past 2nd from the current non-finalized Period)

Output: *available_tx* – subset of pending transactions this node has jurisdiction over

```
1:  function FINDTXJURISDICTIONSET (P, N):
2:      jurisdiction_cert ← hash of the proposing node's signature of past_period_block_id
3:      for each transaction in P:
4:          if (transaction modulo N) equals (jurisdiction_cert modulo N) then
5:              add transaction to available_tx
6:      return available_tx
7:  end function
```

Note that the block id of the past Period block and its signature need to be part of the block to act as proof to help other nodes to validate whether the correct jurisdiction has been used. For each such jurisdiction proof generated, it will remain valid for two (2) Periods, for the same reason the block generation eligibility ticket expiration to account for fuzzy boundary conditions between Periods due to propagation latency.

Also note that, the implicit definition of work load in this algorithm is simply the transaction count. This is a reasonable measure of load during block generation since, in the Taraxa protocol, blocks on the block DAG are not executed immediately means that the load is purely based on validation, which is a relatively simple and equal workload across coin and smart contract transactions.

In both cases, there is no coordination between the nodes on block proposal eligibility and transaction jurisdiction and a certain amount of tolerance or "fuzziness" is built into the validation, thereby reducing the associated network overhead and attack surface.

4.5 Adaptive Protocol

Network conditions are constantly changing, and the rules governing protocol behaviors should likewise adapt – automatically – not via offline meetings, online forums, or instant messaging.

Since Taraxa already uses a period fast PBFT process to confirm Period Blocks, these blocks could easily contain updated parameters based on recent network conditions, parameters such as block generation rate, block size, VRF committee sizes, or even networking protocols. These parameters could all be calculated and determined dynamically on the fly based on real-time network conditions, hence minimizing the need for hard forks and flame wars.

For example, ideally speaking, the expected value of the product of the number of eligible block generators and number of transaction jurisdictions should be ~1. If the product is significantly less than 1, then the network suffers from a high orphan rate. If it is significantly higher than 1, then the network has poor block efficiency. To ensure the network stays at and around optimum performance requires continuous parametric calibration and consensus on these calibrations.

The machine learning algorithms that govern how these parameters are calculated are an ongoing research topic for Taraxa, and we will release our findings as we move forward.

5. Concurrent Smart Contracts

5.1 Brief Background

Brief preface: Taraxa's work on concurrency is inspired by the work of Professor Maurice Herlihy and the pioneering papers [11] [12] on smart contracts and concurrency written under his supervision. Taraxa is fortunate to have Professor Herlihy as our advisor on concurrency and distributed systems technology, and to have had the privilege to collaborate with many of these papers' authors.

Blockchain technology began and was popularized by the arrival of Bitcoin, which primarily served as a way for users to conduct transactions without a centralized third party. As blockchain technologies evolved, an additional layer was added between the client and the underlying ledger, often called smart contracts, which is a set of logic that enables more sophisticated applications to be written on top of the blockchain. One of the earliest and most widely-used implementations of such a smart contract system is a core feature of the Ethereum [6] project. For the purposes of this whitepaper, we will primarily use Ethereum as a point of reference when talking about smart contracts.

A smart contract is similar to an object in a programming language, it has a persistent state (object variables) and a set of functions. Functions could be called directly by a client or by other contracts to alter its own persistent state. In Ethereum, the processing of contracts costs gas, which is paid with ETH, the Ethereum network's native cryptocurrency. The preferred smart contract programming language in Ethereum is Solidity, which compiles into a Turing-Complete bytecode.

In Ethereum, every miner pulls in contract calls (and value transfers), orders them into some sequence, process them, record the resultant new states (locally) and pack the contracts' state transitions into a new block and then publish onto the ledger. Whoever's proposed block ended up on the longest chain wins and earns the gas fees of the contracts and transactions packed into the block.

All smart contract processing is performed sequentially in Ethereum. This is most likely an explicit design choice, as many contract calls, if processed in parallel, will result in conflicting accesses to persistent storage. Also given that Solidity is a Turing-complete language, it is not possible statically determine which smart contract calls will lead to conflicts and hence cannot be avoided ahead of time.

To achieve concurrency, we borrow some techniques from software transactional memory (STM). Most concurrency today is enabled by locking-based techniques, which places a great deal of burden on the developer to make correct design decisions with regards to granularity and minimize lock contention and deadlocks. In fact, this approach towards concurrency tends to become so complex that in practice most developers simply avoid concurrency altogether. STM on the other hand takes the burden off the developer and by optimistically executing transactions in parallel, if a conflict is detected during runtime, the conflicting transaction's changes are rolled back, and the transaction is re-executed at another time. Finalized non-conflicting changes are then committed to persistent shared storage.

Compared to typical lock-based techniques, STM is much more easily adopted by coders as no additional effort is required. It also has the added benefit of being able to create atomic operations that are composable [13], making it much easier to collaborate and compose larger applications from smaller ones in a distributed manner. STM's most obvious drawbacks are the additional coordination overhead incurred by keeping track of shared storage access as well as the cost of rolling back conflicting operations.

In the context of smart contracts, from past studies [11], we see that at relatively low rates of conflict, STM techniques provides significant performance boosting. In Taraxa, we make multiple modifications to existing EVM (we're using EVM as our initial step) as well as simulated financially-incentivized low-contention block packing behaviors to further minimize contention.

5.2 Speculative Execution

As it stands today, all smart contract processes are done in sequential order on a single thread. Here we propose a way to process them in parallel, in order to greatly increase the processing throughput of smart contracts.

There are several obstacles to running smart contracts in parallel. First, because smart contracts modify shared storage (their persistent storage), it's crucial to keep track of which threads are accessing which areas of storage at any given moment to avoid conflicting access. Second, because the programming language is Turing-complete (e.g., Solidity on Ethereum), it is impossible to determine statically whether the different contracts will produce conflict during parallel execution.

We propose that the Taraxa full nodes executes the smart contracts' code as speculative actions. A full node schedules multiple smart contract calls for parallel execution, and then keeps track of their access to persistent storage via the Taraxa runtime APIs. Should there be conflicting access (i.e., read/write, write/write), the access is rejected, the conflict is reported to the scheduler, with the scheduler terminating the thread, rolling back its speculative changes to the persistent storage, and re-schedules these conflicting contract calls for sequential processing.

Note that this is a highly-simplified version of speculative execution, as it involves the wholesale termination of an entire smart contract execution upon encountering any conflict. This approach is a subset of a more generalized approach whereby specific conflicts are tracked within each contract execution and are held for sequential

execution, creating a fork-join schedule that could later be deterministically reproduced. This approach becomes increasingly relevant as smart contracts become increasingly sophisticated and complex.

The next iteration of Taraxa's concurrent VM will implement the more generalized case of speculative execution.

5.3 Concurrent Schedules

Because contracts are now processed in parallel, it's important for every node to follow the same concurrent schedule. However, existing STM systems are non-deterministic, which means that even if several different nodes are given the exact same set of transactions and they speculatively execute them independently, they could end up with different concurrent schedules. Different concurrent execution schedules will often result in different end states, thus breaking the ability for nodes to validate each other's proposed schedules and breaking consensus altogether. To ensure that all executions are perfectly deterministic, the network needs to guarantee that every node follows the exact same concurrent schedule for a given transaction set, which usually comes in the form of a block. This consensus is enforced on the Finalization Chain: right after a Period Block has been finalized, another vote is initiated to reach consensus on the concurrent schedules for every block in the newly-finalized Period, and the resulting set of concurrent schedules are encoded into another block which follows the Period Block finalization block (see Figure 1). This voting process is governed by the same VRF-enabled PBFT process that enables the rapid finalization of the block DAG.

During Concurrent Schedule discovery, a full node selects a set of smart contracts calls it wishes to process and then executes them in parallel, instantiating a separate VM thread for each call. Throughout the processes, it monitors the Conflict Detector for any alerts it may raise (see Figure 2). If it sees a conflict, the Scheduler then terminates the process running the conflicting transactions and rolls back the changes to the State Log (proxy for persistent storage prior to committing the state changes) they've made. It then places the conflicting transactions into the sequential execution queue and executes them after the other parallel processes have completed. Throughout the entire process, smart contracts calls executed in parallel or sequence are recorded into a concurrent schedule S, which is then returned as the Scheduler terminates.

The pseudocode below describes the Concurrent Contract Scheduler run by the representative node to execute contract calls in parallel.

Algorithm 6: ConcurrentContractScheduler(C) – process contracts in parallel

---

Input: A set of smart contracts calls C selected by the representative node

Output: Commit a set of changes to persistent storage and publish a schedule S for validators to follow

```
1:  function ConcurrentContractScheduler(C):
2:      Initialize the storage conflict detector T
3:      Initialize the concurrent schedule log S
4:      Initialize a sequential execution queue F
5:      Execute all contract transactions c ∈ C, watch for alerts from the conflict detector T
6:      if T raises alert of conflict on transaction c:
7:          Roll back transaction c
8:          Place transaction c into F
9:      For each successful thread record the execution sequences into S
10:     For each transaction f ∈ F execute them in sequence and record execution sequence into S
11:     return S
12: end function
```

5.4 Conflict Detector

The Conflict Detector (see Figure 2) tracks all memory access during the speculative executions instantiated by the Scheduler, and reports back conflicts which result in the termination, rollback, and placement of conflicting transactions into a sequential set (see Section 5.3). At the core of the Conflict Detector is a concurrent hashtable data structure which keeps track of all memory being accessed during speculative execution.

Taraxa uses a linearizable set with an insert(key) operation that inserts key and returns True if key was not present in the data structure, and False otherwise. Keys stored in the set are pairs consisting of contract and storage addresses, which is the level of granularity at which data is loaded and stored in persistent storage.

Each node in the set is augmented with two additional fields: state and txn. The insert(key) function is modified to also accept a txn variable as an argument and return a node in addition to the original Boolean value. The returned node is the node found by insert if key was already present, or the newly-created node that now stores key. Each time a transaction txn wants to access a storage location addr, it must make a call to insert(addr, txn). The state variable can be equal to READ, SHARED, or WRITE, and represents which transactions may access the storage location addr. READ denotes a single reader, SHARED denotes multiple readers, and WRITE denotes a single writer. The READ state may change to either SHARED or WRITE, but once changed, it cannot transition to another state. The txn variable is the hash of the transaction that first accessed addr.

The pseudocode below for a conflict detector that uses a modified linearizable set as described above. In the following, CAS refers to the compare-and-swap synchronization primitive. In our implementation, we used a C version of the dynamically-resizable concurrent hash table by Lie, Zhang, and Spear [14], a state-of-the-art data structure. As the original Java implementation depends heavily on the native garbage collector, we use the Boehm-Demers-Weiser garbage collector [15] for our C implementation.

The pseudocode below describe the Conflict Detector used to track access to persistent storage and report conflicts back to the Scheduler.

Algorithm 7: READACCESS(*addr*, *txn*)

Input: Address *addr*, transaction *txn*

Output: Boolean indicating whether there was a conflict

```
1:  function READACCESS(addr, txn):
2:      call insert(addr, txn), which return node node.
3:      if insert succeeded:
4:          return True
5:      read node.txn
6:      if node.txn is equal to txn:
7:          return True
8:      read node.state
9:      if READ:
10:         do CAS(node.state, READ, SHARED)
11:         if node.state is READ
12:             return True
13:         return False
14:     else if SHARED:
15:         return True
16:     return False
```

```
17:  end function
```

Algorithm 8: WRITEACCESS(*addr*, *txn*)

Input: Address *addr*, transaction *txn*

Output: Boolean indicating whether there was a conflict

```
1:   function WRITEACCESS(addr, txn):
2:     call insert(addr, txn), which return node node.
3:     if insert succeeded:
4:       return True
5:     read node.txn
6:     if node.txn is equal to txn:
7:       read node.state
8:       if READ:
9:         do CAS(node.state, READ, WRITE)
10:        if node.state is WRITE:
11:          return True
12:        return False
13:      else if WRITE:
14:        return True
15:      return False
16:    return False
17:  end function
```

6. Technical Roadmap

Light nodes
Protocol-level support for light nodes conducting randomized polling of the network, hardware ASIC to accelerate ECC computations on edge devices.

Concurrent VM
Granular speculative execution with fork-join schedules, smart concurrent objects (e.g., counters) that are contention-aware, non-contentious contract templates, compilation into WebAssembly bytecode, smart contract tagging to make the network aware of intrinsically conflicting contracts, development of a new non-Turing complete programming language that allows for rapid validation without executing the entire contract, integration of off-chain concurrent smart contract processing.

Sidechains
Addition of state-sharding sidechains for individual DApps, cross-chain transactions, automated shuffling of full node validators across sidechains.

7. Economics

The Taraxa token, a native token of the layer 1 Taraxa ledger, is used as a means of exchange as well as fuel in the Taraxa ecosystem and is critical to the operations of the ecosystem. Full nodes will need to be paid a fee to process the smart contracts, and the purchase of IoT data for example is facilitated using the Taraxa token.

A more comprehensive description of the ecosystem's economic design will be published later and its summary will be integrated into this whitepaper.

8. References

[1] K. Granville, "Facebook and Cambridge Analytica: What You Need to Know as Fallout Widens," The New York Times, 19 March 2018. [Online]. Available: https://www.nytimes.com/2018/03/19/technology/facebook-cambridge-analytica-explained.html. [Accessed 15 November 2018].

[2] D. MacMillan and R. McMillan, "Google Exposed User Data, Feared Repercussions of Disclosing to Public," Wall Street Journal, 8 October 2018. [Online]. Available: https://www.wsj.com/articles/google-exposed-user-data-feared-repercussions-of-disclosing-to-public-1539017194. [Accessed 15 November 2018].

[3] EU GDPR.ORG, "GDPR FAQs," EU GDPR.ORG, [Online]. Available: https://eugdpr.org/the-regulation/gdpr-faqs/. [Accessed 15 November 2018].

[4] Y. Sompolinsky and A. Zohar, "Secure High-Rate Transaction Processing in Bitcoin," 31 December 2013. [Online]. Available: https://eprint.iacr.org/2013/881.pdf. [Accessed 30 April 2019].

[5] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 31 October 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf. [Accessed 30 April 2019].

[6] "Ethereum White Paper," [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper.

[7] Y. Sompolinsky and A. Zohar, "PHANTOM, GHOSTDAG: Two Scalable BlockDAG protocols," 2018. [Online]. Available: https://eprint.iacr.org/2018/104.pdf. [Accessed 17 December 2018].

[8] A. Churyumov, "Byteball: A Decentralized System for Storage and Transfer of Value," 1 October 2016. [Online]. Available: https://obyte.org/Byteball.pdf. [Accessed 30 April 2019].

[9] "Bitcoin Confirmation," Bitcoin Wiki, [Online]. Available: https://en.bitcoin.it/wiki/Confirmation. [Accessed 30 April 2019].

[10] J. Chen and S. Micali, "ALGORAND," 26 May 2017. [Online]. Available: https://algorandcom.cdn.prismic.io/algorandcom%2Fece77f38-75b3-44de-bc7f-805f0e53a8d9_theoretical.pdf. [Accessed 30 April 2019].

[11] T. Dickerson, P. Gazzillo, M. Herlihy and E. Koskinen, "Adding concurrency to smart contracts," in *ACM Symposium on Principles of Distributed Computing*, 2017.

[12] V. Saraph and M. Herlihy, "An Empirical Study of Speculative Concurrency in Ethereum Smart Contracts," *arXiv.org*, vol. Computer Science, no. Distributed, Parallel, and Cluster Computing, p. arXiv:1901.01376, 2019.

[13] T. Harris, S. Marlow, S. P. Jones and M. Herlihy, "Composable Memory Transactions," in *PPoPP*, Chicago, 2005.

[14] Y. Liu, K. Zhang and M. Spear, "Dynamic-sized nonblocking hash tables," in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, Paris, 2014.

[15] H.-J. Boehm, "A garbage collector for C and C++," [Online]. Available: http://www.hboehm.info/gc/.