Spring 2019

Randition: Random Blockchain Partitioning for Write Throughput

David Nguyen
San Jose State University

Randition: Random Blockchain Partitioning for Write Throughput

A Thesis

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

by

David Nguyen

May 2019

The Designated Thesis Committee Approves the Thesis Titled

RANDITION: RANDOM BLOCKCHAIN PARTITIONING FOR WRITE THROUGHPUT

by

David Nguyen

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2019

Teng Moh, Ph.D.	Department of Computer Science
Robert Chun, Ph.D.	Department of Computer Science
Suneuy Kim, Ph.D.	Department of Computer Science

ABSTRACT

RANDITION: RANDOM BLOCKCHAIN PARTITIONING FOR WRITE THROUGHPUT

by David Nguyen

This paper proposes to support dynamic runtime partitioning of Tendermint, which is an in-development state machine replication algorithm that uses the blockchain model to provide Byzantine-fault tolerance. We call this variation Randition. We incorporate recent research from blockchain consensus and replicated state machine partitioning to allow Randition users to partition their blockchain for improved write performance at the cost of some Byzantine fault tolerance. We conduct an experiment to compare the raw write throughput of Randition and Tendermint. Finally, we discuss the experiment results and discuss further improvements to Randition.

ACKNOWLEDGMENTS

First and foremost, I want to thank my three committee members. All of their classes inspired this project topic in some way or another. Dr. Chun, Dr. Kim, this project would definitely be lesser without your invaluable input and feedback. I would like to further thank Dr. Teng Moh for his role as my advisor during this project. Dr. Moh, you provided much-needed guidance in times of great self-doubt.

I would also like to extend my thanks and gratitude to my family, friends, and coworkers at IBM. They provided me the moral and emotional support to see this all the way to the end no matter how difficult work-school-life balance became. You know who you are. Thank you all very much.

Special thanks to Frank Butt, lecturer at SJSU and manager at IBM, for helping me get into a career position to make this effort feasible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Introduction

The search for practical consensus algorithms for distributed and replicated state machines continues. As long as we have to account for problems inherent in modern network infrastructure and the mischievousness of human society, distributed consensus remains a topic of research and exploration. Three-Phase Commit was perhaps the first renown improvement over its more naive, two-phase sibling. Enhanced Three-Phase Commit (E3PC) improved upon that. Many variations and new algorithms for distributed consensus have been proposed since. The earliest research focused on tolerating failing nodes. This transitioned into overcoming network failures such as partitioning and message delay with the proliferation of the Internet and today's modern network infrastructure. Paxos and Raft are notable algorithms in this era. Research has recently transitioned into tolerating Byzantine faults with the introduction of blockchain and public distrust of centralized systems.

Past cryptocurrency's promise of decentralized and unregulated money, the research community has realized blockchain can address the Byzantine fault weakness in popular consensus algorithms. The term itself is derived from Lamport, Shostak, and Pease's abstract Byzantine Generals Problem, in which generals of the Byzantine army must decide on one action when some generals are traitors who want to prevent the loyal generals from reaching a consensus [1]. If a system is Byzantine fault tolerant (BFT), it can perform reliably in the face of some malicious nodes actively attempting to stop or fail consensus. Note that this tolerance also covers: failing nodes, network partitions, and message delays that are the result of hardware problems—since one might not be able to distinguish the difference between a hardware failure and a Byzantine party.

A general survey of blockchain technology will reveal that blockchains are essentially distributed databases and replicated state machines. These lend themselves to participation by the general public through numerous blockchain-specific consensus protocols. This explains the terms *distributed* and *public ledger* [2] [3] [4] as common descriptors of blockchain. As of 2019-02-28, Bitcoin transaction throughput has topped at 425,008 daily confirmed transactions according to statistics provided by Blockchain Luxembourg S.A. [5]. That equates to approximately 17,709 transactions per hour and 295 transactions per minute. Median confirmation time for an individual transaction for that month is roughly 10 minutes. Ethereum transaction throughput has topped at 1,349,900 daily confirmed transactions according to statistics provided by BitInfoCharts [6]. That equates to approximately 56,246 transactions per hour and 937 transactions per minute. Median confirmation time for an individual transaction for that month is usually less than 20 seconds. Suffice to say, these performance metrics are outmatched by virtually all conventional database systems—even distributed databases—available now.

Background

Tendermint is a proof-of-stake consensus algorithm that can replicate a state machine while tolerating Byzantine failures in up to one-thirds of nodes or *validators* [7]. Given at least two-thirds are non-Byzantine and correct, the algorithm can guarantee safety, liveness, and even accountability. Validators maintain their own copy of a replicated blockchain and take turns proposing blocks (just large batches of transactions) in rounds. Two voting phases and two well-defined lock rules ensure that no correct validators accidentally fork the blockchain by getting too far ahead of the

consensus, thereby providing safety. Liveness is provided in the unlock conditions that accompany the locking rules and in block proposal timeouts. Byzantine fault testing from the authors and Jepsen—a third party distributed consensus testing firm— [8] confirmed effective Byzantine fault tolerance. The Tendermint team's own performance testing reveals a maximum throughput of approximately ten thousand transactions per second with specific configuration settings.

Tendermint consensus achieves both safety and liveness with rather simple rules. Again, two voting phases and two well-defined lock rules ensure that no correct validators accidentally fork the blockchain and always commit the same block at the same height. The pre-vote and pre-commit are the two voting stages [7]. The pre-vote is for preparing the network to take action regarding a block proposal. If a validator pre-votes for a block, then it is indicating the block is valid and that the network should prepare to commit it. Otherwise if a validator pre-votes *nil*, then it is indicating that the network should prepare to move onto the next round, perhaps because a block proposal it received is invalid or one never arrived. The pre-commit is for voting to fully commit a block. If a validator votes nil in this stage, then it is indicating that the network should move onto the next round. If a validator votes for a block in this pre-commit stage and the block does not get committed for whatever reason, then the validator is considered *locked* on that block. The prevote-the-lock and unlock-on-polka are the two lock rules. Note that a *polka* is defined as two-thirds consensus. The *prevote-the-lock* rule is defined as "a validator must pre-vote for the block they are locked on, and propose it if they are the proposer" [7]. This prevents validators in a network from committing different blocks at the same height and guarantees safety. The *unlock-on-polka* rule is defined as "a validator may only release a lock after seeing a polka at a round greater

than that at which it locked" [7]. This prevents validators from being perpetually locked on a block and guarantees liveness.

In general, a Tendermint network attempts to commit a block in every *round*. If a round results in a successful block commit, then that committed block constitutes a new *height* of the underlying blockchain. Each round consists of the following phases: propose, pre-vote, pre-commit, and commit. Of course, a new round can be triggered from any phase as soon as consensus failure or timeout is detected. We leave detailed explanation of how a single round is processed to [7]. However, we wish to note that at every round, the block "proposer is chosen by a deterministic and non-choking round robin selection algorithm that selects proposers in proportion to their voting power." [9]. The following is a simplified explanation of this algorithm: at the beginning of each round, all validators move forward in a priority queue by an amount equal to their voting power; the validator with the highest priority is selected as the block proposer and it then moves backward in the queue by an amount equal to the total voting power of the network. Of course, additional logic handles edge and error cases. A formal proof for this selection algorithm is not yet available, but public testing does indicate the algorithm is indeed deterministic, fair, and Byzantine-fault tolerant.

Algorand's BA* is another consensus algorithm that bears close resemblance to Tendermint. To clarify, Algorand refers to a cryptocurrency and BA* refers to Algorand's consensus algorithm. Algorand tolerates up to one-thirds Byzantine nodes, nodes each maintain their own blockchain copy, and nodes propose blocks of transactions for the blockchain. However, BA*'s proposer selection and voting phases are distinct. The author utilizes a *cryptographic sortition* algorithm that uses distributed verifiable random functions (VRFs) to randomly determine one proposer out of all eligible nodes [10]. Once

the proposer proposes a block, the same sortition algorithm is utilized to randomly determine voting committees of tunable size, which proceed to perform one phase of voting. There is a minimum of four of these repeated voting committees. Given an Algorand network where up to one-thirds of nodes are Byzantine, there is the negligibly small possibility of a fork if each phase repeatedly chooses voting committees that are compromised by Byzantine nodes. BA* includes logic to reach consensus on a single fork to maintain safety and liveness. Because of this consensus scheme, Algorand easily scales to tens of thousands of nodes and hundreds of transactions per second in testing.

Much blockchain research is devoted to private and permissioned blockchains for business and enterprise use cases. However, researchers do concede transaction performance of popular private and permissioned blockchains often leave much to be desired. Frameworks like Blockbench are being developed to provide potential users with valuable benchmark and analysis of private blockchain systems. Following an analysis of experiment results, the authors of Blockbench suggest that blockchain developers can apply design principles from existing proven database systems to improve blockchain. One notable suggestion is to implement *sharding* or *partitioning* to reduce computational costs and improve transaction processing speeds [11]. Enterprise relational databases such as those from IBM [12], Oracle [13], and Microsoft [14] have long used *data partitioning* to great effect in significantly improving overall transactional query performance for workloads on large stores of data. Users essentially use *partitioning keys* to instruct their database to deterministically divide individual data records onto discrete partitions, which can usually be defined on any combination of discrete storage volumes. This tends to increase read performance because database optimizers can take advantage of data partition awareness to easily “prune” their query

plan. This tends to increase write performance because the database can distribute storage write operations across physical storage and essentially parallelize expensive storage drive writes.

As such, we can consider the cost of writing data to physical storage to be a major bottleneck in transaction processing performance. However, the same bottleneck generally does not apply for current blockchain solutions. Where storage read and write are expensive for relational or other classical databases, consensus is often the expensive operation and bottleneck for blockchains [11]. Conveniently, data partitioning a blockchain can also provide some of the performance benefits we observe in more classical database systems. Before blockchain, distributed state machine replication (SMR) did not scale well mainly because of synchronization (or consensus) overhead. Each node had to receive and execute all transactions. Nogueira, Casimiro, and Bessani observe that partitioning such systems into shards can achieve scalability, but note that most implementations that use partitioning don't do so in a very elastic manner [15]. They propose a modular partition transfer protocol to allow node groups to split and merge with minimal impact on performance. This protocol for elastic state machine replication basically involves splitting a node group into two or more sub-groups, where each partition is tasked with ensuring synchronization until they can act independently and the entire system correctly routes transactions to relevant partitions.

Dasu, Kanza, and Srivastava from AT&T Labs-Research propose a compelling and practical scheme of blockchain partitioning into *sub-chains*. They define Proof-of-Location as a mechanism to achieve distributed consensus, where a location certificate proves the geospatial location of a node to a network that potentially spans the planet [16]. They also posit that location-based partitioning is practical for blockchains whose

purpose is to store and transfer coins of monetary value, because users are likely to send such coins to nearby users and rarely to those located far away. This blockchain partitioning scheme has the likely side effect of increased transaction throughput, since network delays are significantly minimized and sub-chains can commit blocks in parallel. This particular work stops short of implementing an example blockchain and presenting performance metrics.

Goals and Hypothesis

We hypothesize that Tendermint's primary transaction processing performance bottleneck is in its consensus. As more validators join a Tendermint network, we surmise it generally takes more time to reach two-thirds consensus which can contend with relevant consensus timeouts the user defines at network configuration time. In other words, larger network sizes impose greater requirements to reach consensus. This bottleneck results in limitations on practical network sizes. Tendermint's creators only test their system with a maximum of 64 validators and test results indicate it only reaches ten thousand write transactions per second when less than 16 validators are participating in consensus [7]. Tendermint supports adding validators while the system is running, but this only increases average consensus time and degrades transaction throughput. We believe write throughput maximum is improved if Tendermint takes advantage of the concepts presented in the partition transfer protocol [15]. We propose a Tendermint variant that: borrows from this partition transfer protocol, defines how it does *partition formation*, takes advantage of Algorand's cryptographic sortition algorithm, and specializes in partitionable workloads to provide significant transaction and network scalability.

We call this variant *Randition*.

We understand this involves some safety compromises. First, allowing Tendermint's validator network to become partitioned means the entire system's Byzantine-fault tolerance is based off the smallest partition's validator count. We believe this is an acceptable compromise and that users must decide if they will tolerate 10 Byzantine validators in a 64-validator network that's partitioned down the middle or 21 Byzantine validators in a single-partition 64-validator network. Second, supporting partitioning means allowing a sort of network partition to occur, which completely compromises safety if a Byzantine validator is allowed to propose the partitioning scheme. We resolve this issue by further proposing that Randition implement Algorand's cryptographic sortition algorithm [10] to allow the network to autonomously and safely partition itself.

Figure 1 describes our high-level architecture. Randition will implement partition formation logic within Tendermint round processing to allow the blockchain to randomly partition. This formation logic will take advantage of Algorand's cryptographic sortition for safe partition formation. Discussions of safely and liveness in [15] shall guide us on the correctness of our partitioning scheme and logic.

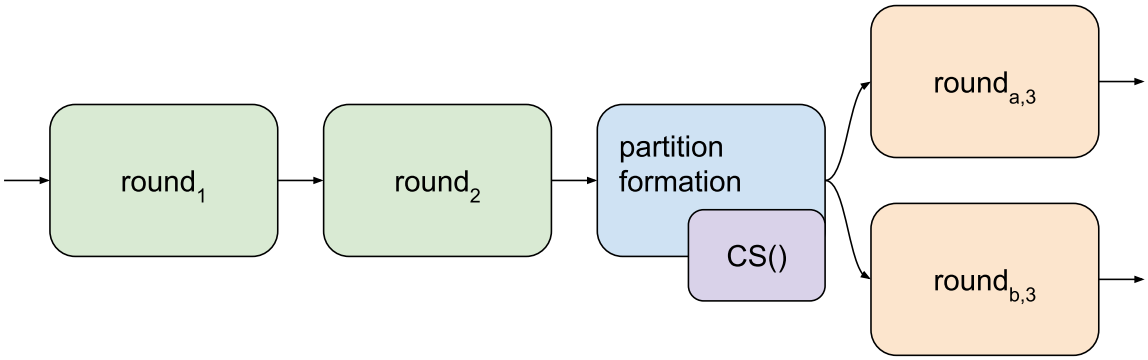

Figure 1: High-level architecture of Randition's partition formation

We wish to distinguish this partitioning operation from the common *network partition* or *blockchain fork* terms in computer science research. These terms possess negative connotation and suggest a severe failure in a network or blockchain. On the contrary, this project intends to improve performance in a Randition network by carefully partitioning the blockchain into sub-chains that can safely process independent tasks. Similarly, we intend to safely partition the blockchain to improve performance for use cases where it is acceptable (e.g. outside of blockchains facilitating money generation and transfers, such as in Bitcoin).

This effort focuses on proving that blockchain write performance is significantly improved when purposefully partitioning a blockchain. Therefore, our specific goal is to observe notable improvement in mean transactions committed per second and mean blocks committed per second. Furthermore, we do not: measure mean transaction commit time or validate transaction linearization. We merely focus on Randition's ability to *commit as many transactions as possible in a time frame* instead of *what it commits within a time frame*. The prototype will not have any cross-partition read capability, but we do discuss how this might be implemented in future iterations. Safe partitioning of the blockchain is also a priority, but we leave proving the safety of our partitioning scheme in future iterations of our work.

Assumptions

To maintain a reasonable scope of work, we make the following assumptions about the expected use cases for our Randition prototype.

Randition assumes that users will initialize the network with each validator having equal or *approximately* equal voting power. Our integration of cryptographic sortition into

Tendermint has dependencies on validator voting power. For simplicity, we discuss our changes and prototype tests assuming each validator has equal voting power. We later discuss the implications of widely varying voting power, but do not perform tests on such possibilities.

Tendermint allows users to define some *peers* with a voting power of zero, which will only allow the peer to observe consensus [17]. For more details, refer to the *Tendermint* section. Non-validator peers only expect to observe consensus between all the validators in a network and will not readily support observation of two or more sub-networks with each performing their own consensus. We discuss the possible usefulness of peers in the presence of multiple partitions, but do not intend to support this and test such a configuration. Further, Tendermint allows users to add peers (including validators) during runtime. However, given the aforementioned assumptions, Randition also does not support the runtime addition of peers to a network or sub-networks. We discuss the usefulness of runtime peer addition in *partition transfer* (see the *Adapted Partition Formation* and *Future Work* sections for more insight), but do not intend to support this and test such an event.

Tendermint possesses some degree of crash recovery—such as the use of write-ahead logging in both the consensus reactor and mempool—to aid in correct state restoration and resumption of consensus participation. Similar to how Tendermint peers do not readily support observation of multiple sub-networks, they will also not readily support resuming consensus with multiple sub-networks following a crash. Minor updates should easily allow a crash-recovering validator to detect which partition they belong to, but we do not intend to support and test this. Of course, crashing a peer is well within the abilities of a Byzantine entity, but we expect Tendermint's existing

methodology for recovering from crashes to be sufficient protection and coverage. Additionally, we consider the safety of crash recovery to be orthogonal to that of partition formation.

Our partition formation protocol currently requires 100% consensus, which means every validator must participate in partition formation within a network-wide time frame and time out. Of course, this is counter-intuitive in our understanding of how blockchains do and should work: consensus should be achieved with a majority of votes instead of all of the votes. However, we put forth the notion that this is acceptable in a permissioned or private blockchain environment where initial network configuration is performed by a trusted user who also controls runtime network *reconfigurations*. In that context, consider that networks should perhaps only partition when this trusted user commands that *all* validators attempt to form partitions at the same time. For users who desire majority consensus and more flexibility, we discuss an alternative partition formation scheme in the *Future Work* section.

Overview

This section begins by discussing the relevant technical aspects of Algorand's cryptographic sortition and partition transfer in elastic state machine replication. It ends by detailing the implementation of these algorithms in Randition and other supporting modifications.

Cryptographic Sortition

Verifiable random functions (VRFs) [18] are an essential component to Algorand's cryptographic sortition. Given a private key *sk* and value *v* as input, function *F* outputs a

seed *x* and proof_x. [18] proves that seed *x* is pseudo-random and unpredictable. Anyone who knows *sk*'s public key *pk* can utilize verification function *V* (consider this an inverse operation of function *F*) with input *pk*, seed *x*, proof_x, and value *v* to determine if seed *x* is indeed valid.

Algorand developed the cryptographic sortition algorithm to randomly and non-interactively select users based on their voting power or *weight* [10]. In order for Algorand to support this algorithm, a pseudo-random seed value must be maintained in the blockchain. Before Algorand begins, initial seed_0 must be selected either by a trusted agent or distributed random number generator and known to all users. At every subsequent round, each block proposer (also known as user *u*) computes the seed for round *r* as follows:

$<\text{seed}_r, \pi> \leftarrow \text{VRF}_{sku}(\text{seed}_{r-1}||r)$, where π is proof of seed_r's validity and VRF_{sku} represents the execution of the VRF with user *u*'s secret key.

Equation 1 [10]

Of course, proposer *u*'s public key is known by every other user. Proposer *u* includes the output seed_r and proof π in its block proposal. As part of block validation, all other users validate *u*'s seed_r with π. If the block is approved for commit, the block is appended to the blockchain along with seed_r. However, if the network deems seed_r from proposer *u* invalid, then the remaining users will compute the seed for round *r* as follows:

$\text{seed}_r = \text{hash}(\text{seed}_{r-1}||r)$, where *hash* is a cryptographic hash function.

Equation 2 [10]

Since the hash function must be identical across the network, the same seed will be computed. Furthermore, since all correct users must have identical copies of the blockchain, all users will have access to the same seed.

Figure 2 summarizes the sortition algorithm [10]. In short, a pseudo-random *hash* is produced from the blockchain's public *seed* concatenated with the desired user *role*, such as the proposer role or a voting committee identifier. Probability *p* is defined as the desired number of users τ out of the total weight *W* between all the users and is the probability an individual unit of weight or voting power is selected by the algorithm—more on this later. A pseudo-random number in the interval [0, 1) is calculated by dividing the *hash* by the maximum numerical value of the *hash*. The interval [0, 1) is also divided into *w* = *j* consecutive intervals as defined by the given cumulative distribution functions. The algorithm checks if the pseudo-random number falls within the defined intervals in order. As soon as the number does, the algorithm returns the amount of weight or voting power selected. Naturally, the first of the consecutive intervals is very large and has a high probability of causing the algorithm to return *j* = *0*. Each following interval is smaller than the last, causing the algorithm to return a high *j* with low probability.

procedure Sortition$(sk, seed, \tau, role, w, W)$:

$\langle hash, \pi \rangle \leftarrow \mathrm{VRF}_{sk}(seed||role)$

$p \leftarrow \frac{\tau}{W}$

$j \leftarrow 0$

while $\frac{hash}{2^{hashlen}} \notin \left[\sum_{k=0}^{j} B(k;w,p), \sum_{k=0}^{j+1} B(k;w,p)\right)$ **do**

  $j{+}{+}$

return $\langle hash, \pi, j \rangle$

Figure 2: The cryptographic sortition algorithm [10]. Licensed under CC BY 4.0.

Let us explain and clarify probability *p* with an example: suppose in a network we have 8 nodes (n = 8) with 10 weight each (w = 10). This equates to a total network

weight of 80 (W = 80). If we want to randomly select approximately 4 out of the 8 nodes, at minimum we want cryptographic sortition to reveal the winners by selecting approximately 1 individual weight (the term *coin* makes more sense in this context) per desired node. This means: 4 weight / 80 weight—or rather: 4 nodes / 80 weight—equaling the desired number of users out of the total network weight. In this case, p = 0.05. Since sortition might not precisely select a desired number of nodes, additional logic such as repeated executions might be required.

Figure 3 summarizes the sortition verification algorithm [10] and is essentially the inverse of the sortition algorithm. If a node is selected and publicly broadcasts the results of their sortition, anyone can verify the results with this algorithm. Cumulatively, the sortition and verification algorithms are essential in providing safe and deterministic selection of committees.

procedure VerifySort($pk, hash, \pi, seed, \tau, role, w, W$):
if $\neg VerifyVRF_{pk}(hash, \pi, seed||role)$ **then return** 0;
$p \leftarrow \frac{\tau}{W}$
$j \leftarrow 0$
while $\frac{hash}{2^{hashlen}} \notin \left[\sum_{k=0}^{j} B(k;w,p), \sum_{k=0}^{j+1} B(k;w,p)\right)$ **do**
  j++
return j

Figure 3: The cryptographic sortition verification algorithm [10]. Licensed under CC BY 4.0.

Partition Transfer

[15] iterates that modern state machine replication primarily provides fault tolerance at the cost of scalability and adding replicas tends to exacerbate this tradeoff. Although recent research already propose partitioning to improve scalability, these tend to not be

very *elastic* in that they don't excel at dynamically partitioning at runtime. Nogueira, Casimiro, and Bessani propose a modular *partition transfer* primitive and protocol in the state machine replication model. Their objective is to enable most state machine replication protocols to implement partitioning with minimal impact to performance and minimal requirements from the SMR protocol. Their protocol can be summarized in Figure 4 [15] and the following 6 steps:

1. Group *G* receives a partition transfer request from a trusted agent.
2. Each replica in *G* sends its state *S* to a matching replica in new group *L*. During this entire stage, updates on *S* are logged in cache Δ.
3. Each replica in group *L* accepts state *S* when it receives and verifies matching *S*-hashes from enough replicas in group *G*.
4. Each replica in *G* sends its cache Δ to its matching replica in group *L*. From now on, group *G* stops serving requests that *L* should be handling and redirects such requests to *L* instead.
5. Each replica in group *L* accepts cache Δ when it receives and verifies matching Δ-hashes from enough replicas in group *G*.
6. When group *G* receives enough acknowledgement messages from group *L*, it reports the partition transfer request results back to the trusted agent.

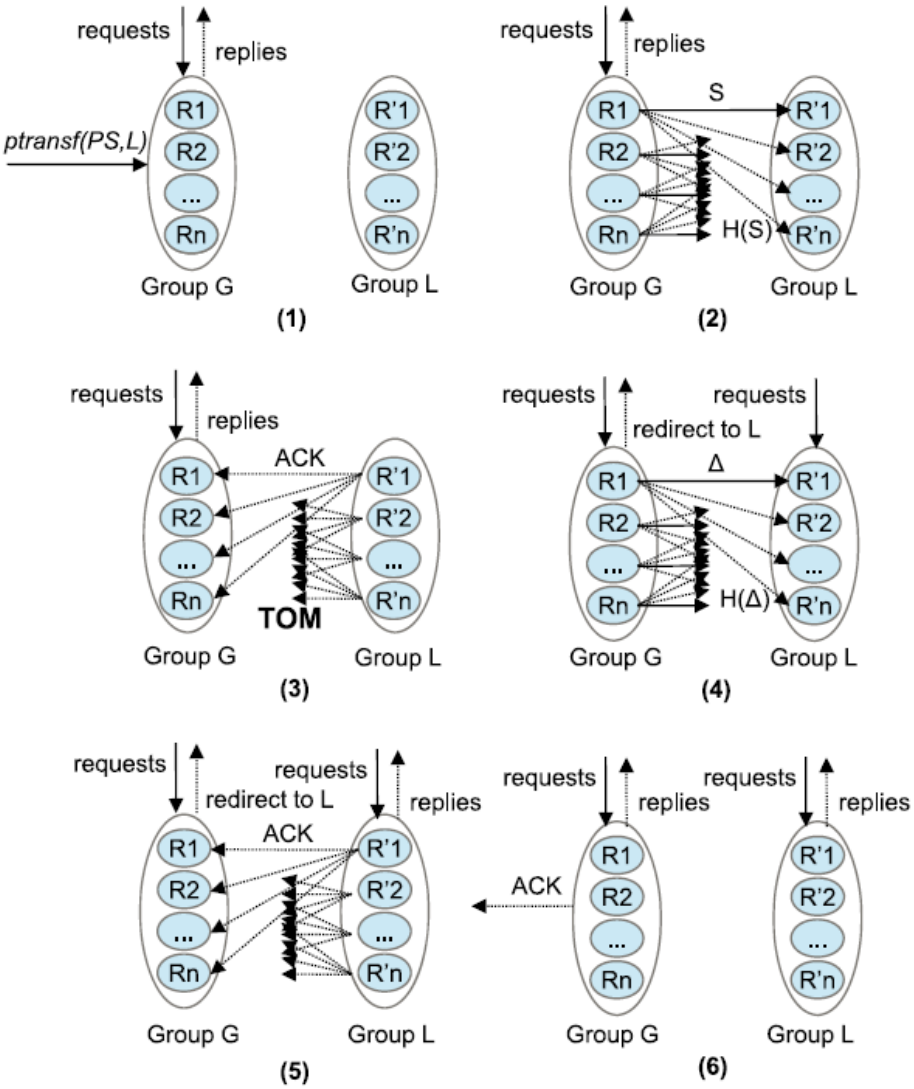

Figure 4: The partition transfer protocol [15]. Reprinted with permission from the authors.

The protocol notes that consensus is only required at the beginning of steps 1 and 4. In step 1, consensus is required for all replicas to start partitioning in a synchronized manner. In step 4, consensus is required for the replicas in group *G* to stop processing requests for the transferred partition in a synchronized manner.

Tendermint

Tendermint promises to never fork as long as less than one-thirds of validators are Byzantine [7]. Our proposal represents a significant change to Tendermint's core processing to allow an intentional fork or partition on our terms. We also contended with the beta state of Tendermint and the volatility of its source code. While Tendermint's core design remains the same, observation of Tendermint's code repository [9] and public documentation [17] over time indicate the entire product is in a constant state of improvement and refactor. Modifying a specific version of Tendermint alleviated most relevant problems. As such, we only worked with and discuss Tendermint version 0.24.0.

From a user standpoint, Tendermint initialization involves using the *init* command to initialize the required files on disk for an individual node, which includes the: *config*, *private validator*, and *genesis* files [17]. The private validator file contains the node's public-private key pair. Users need to modify the config and genesis files to include the entire network's IP addresses and public keys, respectively. Once the network starts, nodes will attempt to dial and gossip with their given peers and verify messages with their known peer public keys. Tendermint distinguishes between *peers* and *validators*. Peers are simply the terminology for a node participating in the network and validators are simply peers with voting power. This means non-validator peers: have zero voting power, can only observe, and keep up with consensus. After initialization, user applications can use Tendermint's API(s) to send transactions and replicate. The *EndBlock* request is one such interface that allows user applications to execute logic at the end of every block commit.

Tendermint has a notion of *reactors*, which are concurrent processes that run alongside the main process and are responsible with helping the validator participate in the network [7]. It does so in part by utilizing a *switch* object to broadcast messages to the entire network, thereby generating *gossip*. The main reactors are the *blockchain*, *consensus*, and *mempool* reactors. The blockchain reactor is responsible for fast-synching the validator with the rest of the network when it is behind on consensus. The consensus reactor is responsible for participating in consensus, such as by gossiping votes and committing blocks. The mempool reactor is responsible for caching, verifying, and gossiping application transactions in the mempool. The mempool can be considered a cache for transactions that have not been committed in a block.

It is important to note that while a transaction is in a validator's mempool, queries from any validator in the network will not observe the results of this transaction. Therefore, Tendermint does not employ an eventual consistency model. Rather, it employs what we consider a *mostly* strong consistency model. Let us clarify. All writes in Tendermint must be completed through transactions, which require consensus to commit. Here, there is strong consistency in data writes. On the other hand, reads in Tendermint are not required to be completed through transactions. Users can simply query the local validator without consensus or other serialization. This method is computationally cheap and quick, but might return stale data [7]. However, users have the option of performing queries as transactions to avoid reading stale data and ensure they're seeing the latest state. The downside is users might have to wait on the order of hundreds of milliseconds to seconds for a response. Here, the user can choose between strong consistency and "slightly weaker" consistency in data reads depending on their requirements.

Tendermint's *Validator Set Update* protocol [7] [17] is integral to our effort. It is triggered when the user application specifies a new validator set at the EndBlock request. Adding a new validator is simply a matter of providing the new validator's public key and desired voting power to the current set. Updating an existing validator is a matter of specifying the desired voting power. Removing a validator simply requires specifying a voting power of 0. Note that updates specified in height H will take effect in height H+2.

Adapted Cryptographic Sortition

We modify Tendermint's consensus reactor and engine to maintain a pseudo-random seed value at every height in the blockchain. We adapted Algorand's algorithm for maintaining such a seed with the following primary differences:

- In Algorand, a hash function is used to determine a new seed if the network doesn't accept the proposer's seed. In Randition, the network continues moving onto the next proposer until a valid seed is proposed.
- In Algorand, a new seed can be determined in each round. In Randition, a new seed is determined in every block or height.

When it is time for a Proposer to propose a block, it executes Equation 1 to obtain a candidate seed and seed proof. These are included in the proposal and the proposal is broadcasted to the network for voting. Randition considers the proposal's seed and seed proof as integral to the validity of the proposal itself. Therefore, if the network determines the seed and proof as invalid, then the network will vote against the proposal and the next Proposer will be selected.

To facilitate and easily manage blockchain partitioning, we define a new *partition reactor* in Randition that is responsible for the following:

- tracking the validator's partition and partition status
- tracking the network's partitions and their status
- using sortition to attempt to form partitions

We found it necessary to define *partition phases* so that the main process and various reactors can determine what stage of partitioning a validator is in. The following phases are numbered and are set depending on the following conditions:

0. The default phase; the validator is not in a partition and partitioning has not been requested.
1. The validator's user application has requested partitioning and the validator is waiting for the required sortition messages to arrive.
2. The validator has received all required sortition messages and partitions have formed.
3. The validator has removed all extra-partition peer validators from its peer list.
4. The validator and network is fully partitioned and partitioning is active.

Phase 0 can be considered the default state. When a network is initialized and started, all validators begin in this state. Phase 1 is primarily used by the main process to determine if a new partitioning key should be accepted or ignored. Phase 2 is primarily used by the partition reactor to track the state of partition formation. Phase 3 is used by the main process to track the state of partition formation. Phase 4 is when a validator can consider itself fully partitioned. Tendermint's Mempool and mempool

reactor depend on this phase to determine if transactions should be categorized as intra- or extra-partition. Figure 5 illustrates the valid transitions between the phases.

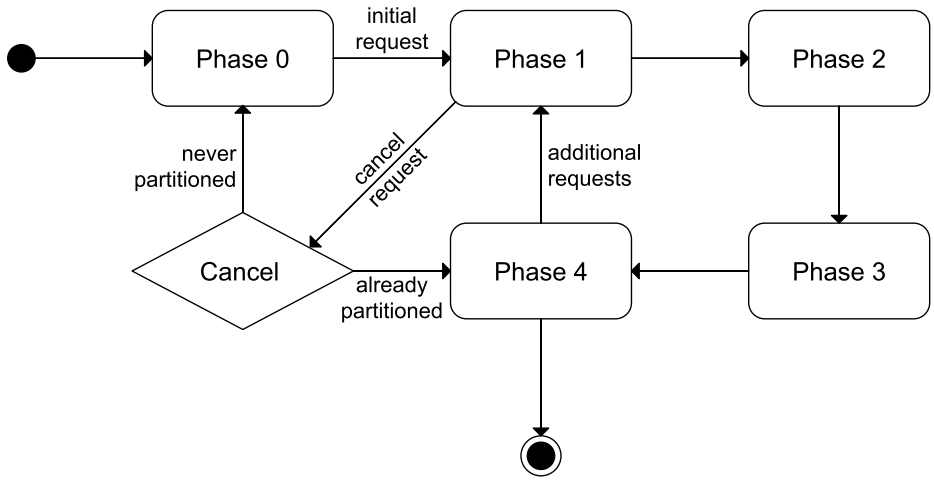

Figure 5: Partition reactor states

We modify Tendermint's external EndBlock request to allow a user to set a new *PartitionKeys* string array with regular expressions, or *regexes*, that inform Randition which partitions to sort transactions into. Let us consider these partitioning keys synonymous to those found in relational databases that support data partitioning. For example, suppose that we replicate a key-value store with a Randition network and suppose that the user application specifies the following 2 regexes for *PartitionKeys*: *(4=)* and *(5=)*. If the network successfully partitions itself, then one partition will only commit transactions where the key is 4 and the other partition will only commit transactions where the key is 5.

We also modify Tendermint's external *BeginBlock* request with the *Partitioned* boolean to inform the user application if and when partitioning is active. Note that Randition does not inform the user application exactly which partition it is assigned to.

Randition's current design does not require partition awareness from the user application. This results in implications we discuss later.

For every round the *PartitionKeys* array is set, the main process will instruct the partition reactor to attempt to partition the network. Whenever the reactor receives this local partitioning request, it will perform sortition using the procedure described in Figure 2 and broadcast the results to the network. We provide the same input parameters that Algorand does into the procedure except for the *role* parameter, which indicates what proposer slot or voting committee a node is competing for. In Randition, the role parameter indicates which partitioning key a validator wants to partition on. To be specific, the role parameter in Randition is actually a hash of the entire partitioning key to maintain a uniform parameter size.

Whenever the partition reactor receives sortition result broadcasts from peer validators, it will either cache the broadcast if the reactor is not aware of a partitioning request or validate the broadcast using the procedure described in Figure 3 if the reactor is waiting for consensus on partition formation. As soon as any validator has verified sortition results from every other validator (equating to 100% consensus), the reactor forms partitions and instructs the main process which partition the validator is a part of. At this point, the validator communicates mainly with intra-partition validators and partitioning is complete—at least until the *PartitionKeys* field is set to request partitioning again. Note that since total consensus is required, user applications should request partitioning at roughly the same time or block height.

Randition uses cryptographic sortition to partition a network into sub-networks while Algorand uses it to select small committees. In Algorand's case, there are "winners" of

sortition and the prize is getting to propose a block, vote on a block, etcetera. In Randition, what prizes do winners get? Presumably, there shouldn't be any *better* partition. We use cryptographic sortition for individual validators to randomly and non-interactively determine their place in a priority queue. From there, Randition can more-or-less evenly divide validators into partitions by picking from the front of the queue and assigning them in round-robin order. Recall that cryptographic sortition selects users based on their voting power. Therefore, validators with more voting power will have a higher chance of being closer to the front of this queue. Figure 6 describes our current partition formation scheme:

```
procedure formPartition(sortitionResults)
  sortByTimesSelected(sortitionResults)
  i ← 1
  while validator ∈ sortitionResults do
    if i % 2 == 0 then partition1Vals ← validator
    else partition2Vals ← validator
  partitions ← < partition1Vals, partition2Vals >
  return partitions
```

Figure 6: The partition formation scheme

Note that *sortByTimesSelected()* breaks ties on times selected with the higher hash sorting higher. Note that formation into only 2 partitions is supported at this time, although the scheme could easily be improved to support a dynamic amount of desired partitions.

The novel use of cryptographic sortition in partition formation is that each validator forms partitions independently from each other following one phase of sortition message exchange with every other validator. Consider a Byzantine validator who wishes to trick another validator into joining the wrong partition or multiple partitions. By the time this Byzantine validator determines which validators belong to which partitions: the partition

formation is complete, there are no additional phases or message exchanges, and no chances to influence partitioning. Of course, a Byzantine validator can completely stop partition formation from occurring with our current 100% consensus requirement by not sending any messages at all, but the key here is that they cannot corrupt partition formation and configuration.

To reiterate, Tendermint is a state machine replication system. User applications need not concern themselves with how many other replicas exist. Users can reasonably assume all correct validators have the latest identical state. However, with our proposal to allow partitioning we potentially break user and application assumptions. Partitioning could mean user applications must be aware that sub-networks exist. It could also mean users must concern themselves with how many states there are and which validators house which transaction partition. We develop Randition with every intention to make partitioning and partition states as abstract as possible. For example, we intend for the *PartitionKeys* field to be a user request for better performance at the cost of Byzantine-fault tolerance rather than a request that requires further action or knowledge on the user or application's part.

Adapted Partition Formation

Randition is very much inspired by [15]'s proposal and we don't precisely implement their defined protocol. Instead, we rely on it to provide valuable guidance and insight towards maintaining safety and liveness during blockchain partitioning. For example, partition transfer specifies "having one single replica in the source group transmit to one single replica in the destination group [because] this design option makes our system bandwidth-efficient in multi-rack and virtualized environments" [15]. However, individual

Tendermint validators communicate via broadcasts to the entire network instead. We choose to preserve and take advantage of this gossip mechanism instead of implementing pairwise communication.

Step 1 of the partition transfer protocol requires consensus to begin the partitioning process and Randition defines consensus to be a partition request from every validator in the network or a sub-network. In this context, the replicated user applications are collectively the trusted agent. We consider the user application to be trustworthy because Tendermint's authors state "transaction verification is the responsibility of the service that is being replicated" [19]. Randition could potentially require only two-thirds consensus instead of 100% consensus and this is discussed later.

Steps 2 to 5 of the partition transfer protocol assume that group L is not state synchronized with group G and might be new additions to the whole network. However, our adapted cryptographic sortition will determine groups G and L from the network's existing nodes. Since the network should always be consistent before and during the partitioning process, these four steps aren't necessary. Suppose a scenario where a user doubles the size of a blockchain network at runtime, one can consider this effectively introducing a group L that is not synchronized with the existing nodes in the network. At the moment, Randition does not support runtime addition of peers during partitioning. We believe an equivalent alternative would be to utilize Tendermint's Validator Set Update and *Fast Sync* protocols to synchronize new group L with existing group G and then initiating partitioning requests network-wide.

Step 6 defines the success condition for the partition transfer protocol, which is: the new partition contains common state S, the new partition contains cache Δ, and both

partitions are correctly processing new requests. Randition defines this success similarly: partition formation succeeded and all the partitions are correctly processing new transactions.

Consider the following Randition's version of the partition transfer protocol:

I. Group *G* receives a partition transfer request from a trusted agent.
II. Group *G* collectively and consistently determines new groups L_1 and L_2, known as *partition formation*.
III. Groups L_1 and L_2 cease participating in consensus with each other and continue consensus locally.
IV. Groups L_1 and L_2 report the partition transfer request succeeded to the trusted agent.

[15] defines the requirements for compatibility with their partition protocol and they can be summarized as follows:

1. All system processes communicate messages through *fair channels* and that all messages will eventually be received at the destination, regardless of network problems or delays.
2. The distributed system is partially synchronous where it "can behave asynchronously for an unknown period of time, not respecting any time bound on communication or processing, but eventually will become synchronous" [15].
3. The system possesses an external trusted component that can trigger group reconfiguration operations, such as partitioning.

Tendermint fulfills Requirement 1 because it satisfies *atomic broadcast* for consensus [7], which includes block proposal of data and vote gossip in consensus.

Regarding Requirement 2, Tendermint states it is *weakly synchronous*, meaning that messages for each stage of consensus must be communicated within a user-specified time, otherwise Tendermint moves on to the next stage following this timeout. This definition is stricter than that of partial synchrony [20]. But assuming that Tendermint users specify reasonable timeouts in which the non-faulty majority of validators can eventually finish sending and receiving messages, then we can assume Tendermint fulfills Requirement 2. The latest formal update from the Tendermint team clarifies that their protocol operates in a partially synchronous model [19]. Tendermint fulfills Requirement 3 by trusting the replicated user application to handle group reconfiguration operations, such as usage of the Validator Set Update protocol in Tendermint's EndBlock request interface.

[15] states the partition transfer protocol fulfills safety and liveness in the following ways:

- Safety 1: once partitioning completes, "the transferred partition will not be part of the source group state and will be part of the destination group state" [15]
- Safety 2: "linearizability of the service is preserved by the partition transfer" [15]
- Liveness: partitioning eventually completes

Randition satisfies *Safety 1* due to Steps II and III. Partition formation in Step II ensures two independent partitions and Step III ensures independent "group states" and data consistency because every partition does not process extra-partition requests. Randition satisfies *Safety 2* because Tendermint satisfies *state machine safety* [7] and will continue to do so after partitioning completes due to Step III, where the majority of validators in any partition is expected to continue Tendermint's consensus protocol.

State machine safety guarantees total order for blocks committed and transactions therein.

Randition satisfies *Liveness* because all of the Steps I through IV must terminate. In Step II of our protocol, the partition reactor will only wait as long as a round for all of the required sortition messages to arrive. If they do and they are all valid, partition formation occurs and completes independently. If they do not, then Tendermint carries on and another partition request must be made in a future round. Steps III and IV execute independently of other intra-partition validators. Step III relies on execution of Tendermint's Validator Set Update protocol, which must terminate before a new block is proposed. Step IV is completed once Tendermint responds to its BeginBlock interface, which must occur before each new block commit.

We make the following additional comparisons between the original partition transfer protocol and Randition's partitioning protocol:

- The partition transfer protocol states that consensus is required in Step 1 and 4 to ensure that partition transfer begins consistently for group G and exits consistently for both groups G and L. More specifically, consensus in Step 1 ensures that the majority of group G generates identical state S and cache Δ. Additionally, consensus in Step 3 ensures that the majority of group G: finally sends identical state S, finally sends identical cache Δ, and stop executing extra-partition requests in a synchronized manner. There is a distinct focus on data consistency at these stages.
    - Our partitioning protocol only requires consensus in Step II and prioritizes consistency in partition formation. Randition relies on Tendermint's

existing block consensus mechanism to maintain data consistency and our partition formation processing does not interfere with this mechanism. Before partitioning, Tendermint's consensus protocol promises data consistency. *During* partition formation, Randition continues behaving as though the network is unchanged. The partition reactor independently deals with partition formation and doesn't effect change in the main process or other reactors until formation is finalized and complete. Therefore, data consistency is maintained. Partitioning is activated using Tendermint's Validator Set Update protocol, which only occurs *between* block proposals. After partition activation: a validator only accepts block proposals from intra-partition Proposers, we effectively observe partitions that independently perform block consensus, and data consistency is maintained within each partition.

- The partition transfer protocol makes minor notes in Steps 4 and 5 regarding group *G* ceasing service requests for partition *PS* and redirecting them to group *L* while group *L* waits to finish receiving cache Δ before servicing these *PS*-related requests. These finer detail specifications are for maintaining data consistency and we make the same point as above regarding our focus on consistency in partition formation. We continue to rely on Tendermint's existing block consensus mechanism for data consistency.

Consider Figure 5, where each partition phase is represented as a state in a state machine. This begs the question regarding if state transitions are safe and live. Can a correct validator be coerced into incorrect states or forced to not terminate? Internally, the partition reactor is only concerned with these phases. The rest of Randition—

especially Tendermint's core and mempool reactor—is mainly concerned with whether partitioning is fully activated or not. This results in two similar but separate concerns. One: whether state transitioning is safe and live within the partition reactor between the 5 phases; and two: whether state transitioning is safe and live outside of the partition reactor (e.g. the core and mempool reactor) between partitioning being fully activated or not. Note that Phase 0 and 4 are at-rest states the partition reactor can remain in for an extended period of time. Phases 1 through 3 are transitionary and all of them should ideally be completed within several—if not one or two—rounds.

Both safety concerns are addressed by the definition of Phase 1: *the validator's user application has requested partitioning*. More specifically, a correct validator cannot be coerced to transition outside of Phase 0 by a Byzantine entity. If the validator's user application incorrectly requests partitioning, the validator will trust the user application and the validator itself is no longer correct. We showed that the transition through Phase 2 by a correct validator cannot be corrupted by a Byzantine entity. Phases 3 and 4 are informational and determined by Tendermint's round processing. As long as Tendermint can continue processing rounds, then Phases 3 and 4 are guaranteed to execute correctly.

The primary liveness concern lies in Phase 1, where the validator is waiting for the required sortition messages to arrive. As long as the partition reactor is in Phase 1 and 100% consensus has not been achieved, the partition reactor (and the network as a whole) will continuously execute and gossip sortition for each new round in an attempt to form partitions. We purposefully allow users to constantly retry since requiring 100% consensus within one or two rounds is admittedly a rather lofty expectation. Other than successfully transitioning to Phase 2, the only other way to guarantee termination of

Phase 1 is for the user application to explicitly request cancellation by providing an empty *PartitionKeys* array in the EndBlock request. We emphasize that while the partition reactor remains in Phase 1, Randition is able to continue consensus and round execution, because Randition's core is mainly concerned with whether partitioning is fully activated or not. Phase 2 terminates because partition formation is completed independently of other validators and merely a matter of executing the partition formation scheme indicated in Figure 6. As before, Phases 3 and 4 are informational and determined by round processing. As long as Tendermint's round processing maintains liveness, then Phases 3 and 4 are guaranteed to terminate.

After partitioning is activated, the Mempool begins distinguishing between intra- and extra-partition transactions. This is merely achieved by checking if this partition's partitioning key—which is a compiled regular expression at this point—returns a match in a transaction. Following Tendermint's design, all transactions are cached in the Mempool. However, Randition ensures transactions that are designated extra-partition are never proposed. They are retained only for gossip and gossip optimization. The gossip routine in the mempool reactor will repeatedly read the cache and broadcast each transaction to the entire network. At the end of every block commit, these extra-partition transactions are flushed from the mempool to minimize the performance side effects of a large transaction cache containing never-committed transactions.

Note that Randition does not take additional action for extra-partition data that was already committed before partitioning. Therefore, a validator in a partitioned network will have stale data, as indicated by Figure 7. Given a data partition $\mathrm{P}_{\mathrm{p,v}}$, where p is the partition identifier and v is the data state version, observe that the extra-partition data that a validator possesses will become increasingly stale as more blocks are committed

to each blockchain partition. We leave handling stale data and ensuring correct query output to future work.

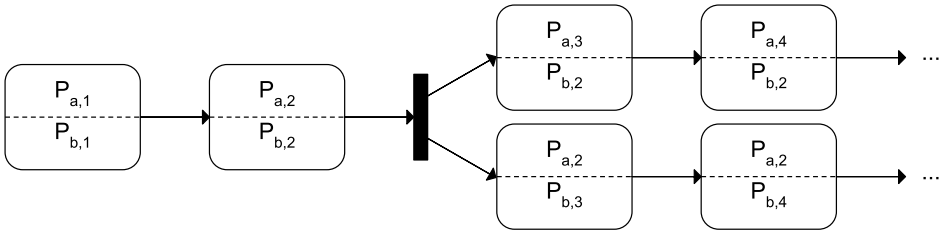

Figure 7: Data partition states

As a clarification, transactions in Randition are gossiped by the mempool reactor normally between all other peers a validator is connected to. What each validator does with a transaction is up to them. However, the same cannot be said for proposal, block, and vote gossip via the consensus reactor. Recall that validators are merely peers with voting power. Therefore, a validator will readily gossip messages with peers, but will only trust messages originating from other validators. In the case where a validator belongs to a partition, it will readily gossip with extra-partition validators, but will only trust messages originating from intra-partition validators. Messages that require verification are signed by the originator and this includes proposals, blocks, and votes. Tendermint's current design will only gossip signed messages after they're cached, and a validator will only cache messages after it ensures the message is trustworthy by verifying the signature or by checking if it expects the message from the originator at all.

Implementation

Tendermint's source code is written in Golang (also known as Go), an open source programming language whose development is led by Google [21]. Of course, our modifications were completed entirely in Golang. Our changes to Tendermint consist of

approximately 1600 lines of code. Randition utilized Coniks' verifiable random function library [22]. Both Tendermint and Coniks use Ed25519 [23] as their public-private key system, but their precise implementations vary enough so that Tendermint keys cannot be used in Coniks and vice versa. As such, each validator is modified to store Coniks-compatible keys for use when calling the Coniks VRF.

Several milestones were defined to create goals of smaller scope and to prompt assessment of the prototype feasibility given code complexity and time constraints. The first major milestone was to partition a simple 8-validator network into 2 predetermined partitions of 4 each. This involved finalization of infrastructure code to initialize the desired number of validators with the correct configuration files. The completion of this milestone was marked by initial definition of the partition reactor and correct execution of partition processing logic in the main execution loop. The second major milestone was to randomize the partitioning. This involved modifying Randition's block processing to include pseudo-random seed generation with Coniks' VRF library. The completion of this milestone was marked by our implementation of cryptographic sortition and achieving identical partition formation across all validators. Functional testing was greatly aided by Tendermint's excellent logging capabilities and conventions.

The third and final major milestone was to demonstrate that randomized partitioning did result in increased write performance. This involved repeated network performance testing against Tendermint and Randition and gathering benchmark data. System testing was greatly aided by Tendermint's benchmark tooling. However, much automation and infrastructure needed to be developed to repeatedly:

1. deploy a large 32-validator network

2. prepare the network for testing by committing a small batch of transactions and/or partitioning the blockchain
3. schedule benchmarks to begin simultaneously on all validators
4. fetch, parse, and archive benchmark results

Results

All experiments occur on Digital Ocean servers of size *s-2vcpu-4gb* (2 vCPU and 4 GB of memory) running 64-bit CentOS 7. All instances are located in Digital Ocean's *SFO2* (West Coast United States) region to minimize the effects of network delay. Each server is initialized using Tendermint-provided Terraform and Ansible scripts for automated network deployment. We modified these scripts to also perform a clock sync and additional Linux package installation to support synchronized testing and benchmarks. We rely on Tendermint's *tm-bench* tool to generate data and benchmark the network. Each validator is configured to execute: Tendermint, Tendermint's example *kvstore* user application, and tm-bench on-demand. All validators are configured to connect directly to each other to minimize the effects of network topology.

Tendermint's main configuration file, the *config.toml* file, is customized by:

- setting the *moniker*, or peer name to a unique identifier
- disabling mempool logging in the *log_level* setting
- disabling empty block commit in the *create_empty_blocks* setting
- significantly increasing the mempool *size* to 250,000
- significantly increasing the mempool *cache_size* to 500,000

We avoid tuning the remaining settings and leave them as defaults to emulate a more *out-of-the-box* and *off-the-shelf* user experience.

Each transaction in our test workload is generated by tm-bench and we selected a transaction size of 500 bytes. Each transaction is an amalgamation of pre-determined and pseudo-random hex-encoded data. New transactions are generated by mutating bytes 16 through 31 of the previous transaction with pre-determined data and by replacing bytes 40 through 89 of that transaction with new pseudo-random hex-encoded data. Our experiments rely on the 81st byte to be pseudo-random during transaction generation, because we modify the example kvstore application to partition with the following regular expression partitioning keys for 2 partitions: *^.{80}[0-7]* and *^.{80}[^0-7]*. These partitioning keys will result in approximately half of the total transactions generated being committed in the first partition and remaining half becoming committed in the second partition.

Repeating experiments consisted of the following workflow:

1. reset the network to restore the original configuration and block height 0
2. *prime* the network with 1 transaction per second for 20 seconds per validator
3. wait 30+ seconds for network activity to settle
4. benchmark the network with the desired transaction rate for 20 seconds per validator
5. fetch the network and tm-bench logs to gather statistics

We are mainly interested in observing transaction throughput *after* the blockchain partitions itself. Therefore, we prime the network with a low transactions-per-second rate to ensure the network is partitioned and that benchmarking does not occur near the

partition boundary or any partitioning processing. We focus on committed transactions per second and committed blocks per second as calculated by tm-bench to be the basis for transaction throughput. For Randition, overall committed transactions per second is defined as the sum of transactions committed per second by all partitions in the network. Since tm-bench might report slightly varying results for each validator in an individual partition, we first calculate the committed transactions per second per partition to be the mean average of intra-partition tm-bench outputs. We use the same definition for committed blocks per second.

Our experiments primarily vary the input transactions per second per validator in a 32-validator Randition network and compare the results to those from a 32-validator Tendermint network on the same workload. We vary the input transactions per second per validator between 25, 50, 75, 100, 150, and 200. That translates to 800, 1600, 2400, 3200, 4800, and 6400 input transactions per second over a whole 32-validator network. Figure 8 compares transactions committed per second between Tendermint and Randition. Note that the error bars indicate the standard deviation.

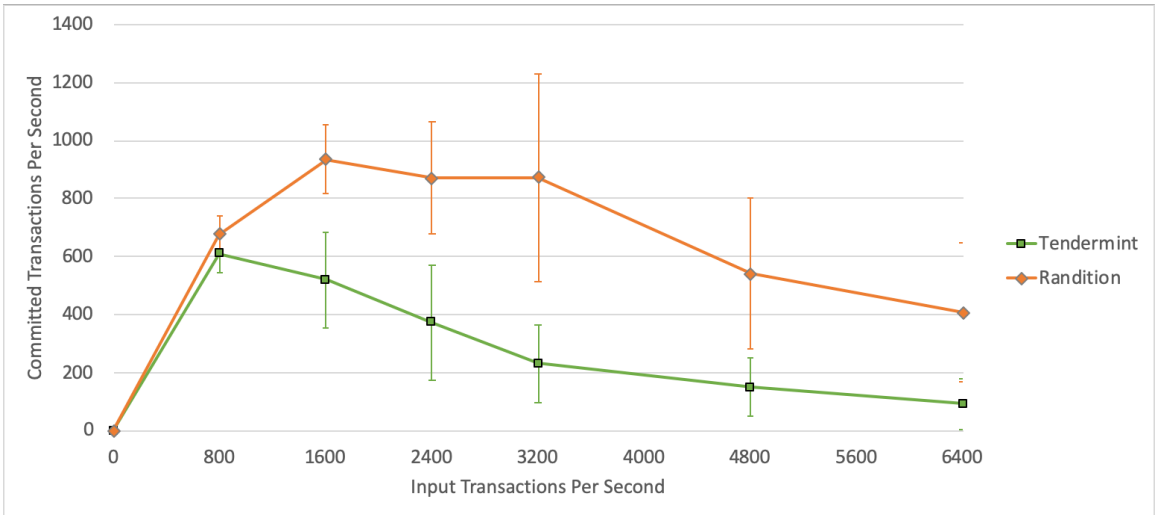

Figure 8: Committed transactions per second

These results are comprised of 12 executions per input rate for Tendermint and 16 executions per input rate for Randition.

Figure 9 compares blocks committed per second between Tendermint and Randition. Note that the error bars indicate the standard deviation.

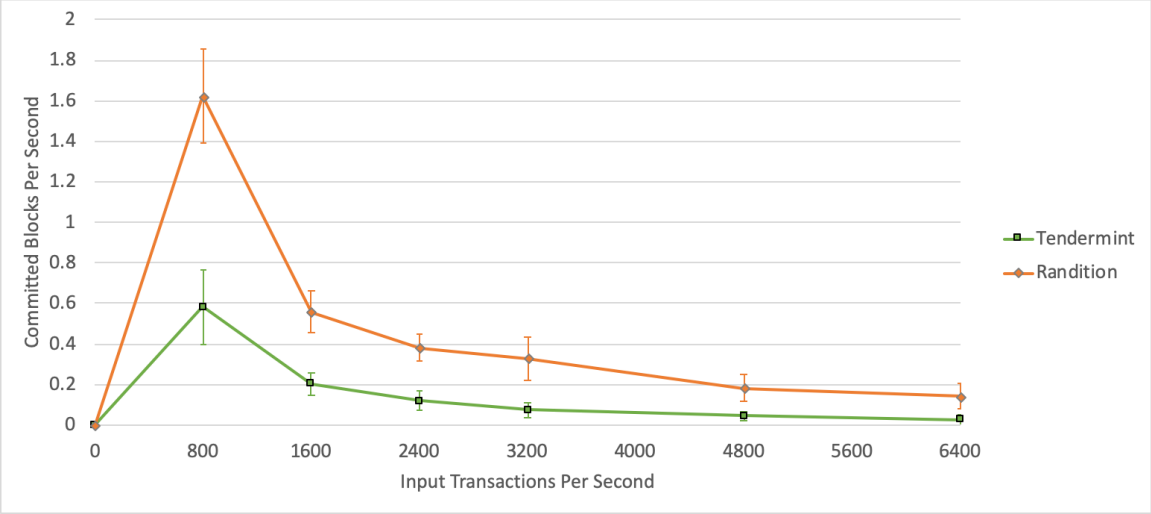

Figure 9: Committed blocks per second

Discussion

To reiterate: this effort focuses on proving that transaction write performance is significantly improved when safely partitioning a blockchain. In Figure 8, we can observe that Tendermint and Randition output similar performance up until approximately 800 input transactions per second for a 32-validator network. However, past a threshold between 800 and 1600 transactions per second, Tendermint's performance begins to degrade and unable to empty the Mempool and commit transactions as quickly as they arrive. Randition reaches its peak committed-transactions-per-second past Tendermint's peak and continues to maintain better performance than that of Tendermint. In Figure 9, we can observe that Randition commits more transactions per second mainly because

the overall network can communicate and commit more blocks in a time frame. Note that by default, Tendermint is configured to process approximately 1 block per second [17] but Randition seemingly exceeds that limitation simply because there are 2 partitions committing blocks independently while all the validators in each obey the same configuration settings.

Since our tests involve virtually the same network configuration and workload generation, we suggest this performance improvement in Randition is a direct result of: blockchain partitioning, dividing the cost of consensus, and parallelizing consensus. We further suggest that the performance bottleneck primarily lies in Tendermint's consensus protocol, at least for a mostly-default network configuration of 32 or more validators. We later discuss the implications of careful tuning and other network configurations. In a 32-validator network, consensus between 2 networks of 16 validators—all of which remain gossiping with each other—is easier to achieve than consensus between 32 validators. To push the example further, it is easier for 2 groups of 16 nodes to independently determine consensus with a two-thirds majority of 11 nodes versus a group of 32 nodes with a two-thirds majority of 22 nodes. Naturally, network gossip and *consensus requirements* will tend to increase non-linearly with more participating nodes. We consider the following part of consensus costs:

- computational processing required for proposals, blocks, and votes
- time required for proposal, block, and vote messages to arrive to achieve consensus

The amount of gossip or gossip overhead also contributes to the improvement in transaction processing performance. Recall that Randition will only gossip messages it

verifies and will not gossip extra-partition proposals, blocks, and votes. On the other hand, Tendermint will gossip these consensus messages between all validators. This translates to a notable reduction in gossip in Randition since each validator will effectively ignore—but still receive—these consensus-related messages from half of the original network. One might consider this a natural optimization of blockchain partitioning, but only if users could tolerate less resilience in the face of Byzantine network partitions. Of course, Randition could be modified to gossip these extra-partition messages to “restore” and emulate Tendermint’s gossip scheme, but we leave this to future work and analysis.

As an aside, Tendermint supports transaction *rechecking*, which will revalidate all remaining transactions in the mempool against the newest block state following a commit [17]. Rechecking is a fairly expensive operation and tuning guides suggest this feature be disabled for workloads that don’t require transaction revalidation. In scenarios where this revalidation becomes the transaction throughput bottleneck, partitioning should alleviate the cost of this operation because Randition does not send extra-partition transactions to be rechecked against the user application. The user might observe more performance benefits if the user application must incur significant delays from operations such as read/write serialization and storage I/O as a result of transaction checking.

One might see that none of our results approach the roughly 10,000 committed-transactions-per-second that Tendermint’s original experiments achieve [7]. However, we note that these numbers were achieved with specific conditions. The authors note that “an artificially high TimeoutPropose is used (10 seconds), and all other timeout parameters are set to 1 millisecond. Additionally, all Mempool activity is disabled (no

gossiping of transactions or rechecking them after commits)" for their first set of raw throughput experiments [7]. We further note that 16-or-more-validator networks struggled to reach this 10,000 transactions-per-second threshold. This reveals that tuning the network will likely result in significant improvements in transaction processing performance and that Randition users should certainly do so in production environments. However, partitioning a network after tuning it is not a focus of this project and we leave such experiments to future work. However, we do expect that users who combine network configuration tuning and partitioning will likely observe excellent transaction write performance.

Of course, we do concede that Randition configuration could potentially be tuned to nullify or worsen the effects of partitioning the blockchain. In general, we expect users of partitionable blockchains to perform partitioning after carefully evaluating potential benefits and costs. We also concede that gossip and network delay could potentially become the primary bottleneck if an extremely large network size is used (Algorand proposes supporting tens of thousands of nodes) and no optimizations are in place within Randition to ignore or manage excessive gossip. In this scenario, blockchain partitioning would be ineffective. We leave further speculation and related experimentation to future work.

While the focus of this project was to demonstrate that blockchain partitioning is certainly viable in improving transaction write performance, we also reiterate that the partition formation scheme is integral in demonstrating that partitioning is actually safe to use and that the network can partition with *safety and liveness*. We have demonstrated in the *Adapted Partition Formation* section that Byzantine validators cannot coerce the network into performing incorrect partition formation or trick a validator into joining the

wrong or multiple partitions. However, other attack vectors are possible outside of our partition formation scheme. Fortunately, Tendermint's modular design and reactor concept aids in minimizing the effectiveness of a Byzantine entity.

Future Work

As we've explicitly stated thus far, Randition requires much improvement and iteration to approach becoming a useful application. This particular effort has focused on developing a prototype to demonstrate feasibility. For example, Randition currently only supports partitioning into 2 partitions *once*. Relatively minor modifications to Randition should allow it to partition into any number of partitions as many times as the user desires as long as Byzantine fault tolerance can be maintained. Recall that crash recovery support also requires minor changes to Randition.

One major functional defect still present in Randition is that the acceptance of the required sortition messages (the transition to Partition Phase 2) is truly single-phase. Recall that as soon as a validator receives all the required sortition messages, it'll move to complete partitioning. However, suppose that a validator in a Randition network somehow doesn't receive the last required message due to a network partition or Byzantine entity. All other validators will move onto partitioning and this remaining validator will assume the network is unchanged. There is currently no mechanism for a validator to persist correct sortition messages for gossip or to request old ones from peers. We commonly encountered this defect during experimentation, but deemed the occurrence was tolerable given the constraints we define early in the *Results* section. Additional work is necessary to support this, and we could take inspiration from Tendermint's Fast Sync feature to assist us.

Increasingly useful and more difficult improvements will involve reevaluations of Randition's ability to maintain safety and liveness. Such improvements follow:

- Support for varying voting power between validators.
- Support for non-validator peers.
- Support for runtime addition of peers following partitioning.

Recall that Algorand's cryptographic sortition selects users based on their voting power and that this effort assumes all validators possess the same voting power. Naturally, given a network with widely varying voting powers, validators will not have the same chance of being selected. However, recall that in our current partition formation scheme the "winners" of sortition merely guarantee themselves close-to-first place in a priority queue to be added round robin into multiple partitions. As long as there are no outliers in a distribution of varying voting power, partition formation should result in partitions with similar cumulative voting power. Additionally, given the random nature of cryptographic sortition, similar cumulative voting power might not be practically achievable with smaller network sizes. We admit that if voting power varies too widely relative to the network size, partitioning is likely not a wise or recommended course of action (as it is with tuning any other setting Tendermint or another blockchain might have).

Previously, we state that Tendermint supports non-validator peers, which can be considered validators with zero voting power. Much design would be required to enable such a peer to observe a network of multiple partitions. It could easily participate in Mempool gossip. However, how would it participate in consensus gossip? Would it be specially configured to trust all validators (via public key) in the network to do so? Which

blocks would this peer commit, if any? And how? On the other hand, relatively little additional work would enable such a peer to observe consensus and participate in gossip with one partition, since a single Randition partition behaves much like a Tendermint network. Overall, the specific role that a non-validator peer would play in Randition currently eludes us. We leave architecture and design of such support to future work.

Remember that Tendermint supports runtime addition of peers (with or without voting power), but Randition does not support such runtime addition during and after partitioning. Currently, the only opportunity for a user to add validators to a Randition network would be to utilize Tendermint's Validator Set Update and Fast Sync protocols to synchronize a new set of validators with the existing set *before any partitioning* and then initiating a partitioning request. Runtime addition of peers following any partitioning requires more modifications. Also recall that Randition never informs the user application what partition it belongs to and the user application is essentially not partition-aware. To add a peer in Tendermint, a user would ensure the new peer's public key and desired voting power is added at the same time across the network via the EndBlock request. However, if we wanted to use the same scheme in Randition, a user would have to ensure the new public key and desired voting power is added at the same time on every peer's user application *in the desired partition*. This would require partition awareness and arguably add unnecessary burden on the user application to handle peer addition in a fully partitioned state. We leave architecture and design of handling peer addition into a partitioned Randition network to future work.

One notable, but minor security defect remains in Randition. Let us suppose a Byzantine entity that is capable of corrupting any number of validators at any time and

directing any of these validators to send any message to other validators. One possible attack vector would be to simply simulate the *PartitionKeys* field being set. This will cause the main process to instruct the partition reactor to attempt partitioning and gossiping sortition result messages. A Byzantine entity could go further and send other validators *fake* sortition result messages that appear to originate from different validators. The partition reactor has a message cache that temporarily stores sortition messages that a validator doesn't expect or cannot verify yet—perhaps because this validator will receive a late partitioning request. Of course, the reactor will verify these messages as soon as partitioning is requested. Currently, there's no signature verification for entry into this cache, so Byzantine entities can denial-of-service (DoS) attack this cache to: prevent or delay partition formation, crash the reactor, and perhaps crash the entire validator itself. We leave resolution of this defect to future work and testing.

Tendermint configuration provides several message-wait timeouts as part of its *weakly synchronous* model. We suggest a new setting named *timeout_partition* in Randition configuration that defines how long a validator waits in Partition Phase 1 (sortition message wait) before automatically cancelling and falling back to Phase 0. Currently, user applications can explicitly cancel partitioning requests to achieve the same effect, but we suggest this setting as an improvement to liveness and to provide users with additional tuning opportunities.

We concede that 100% consensus is an unrealistic expectation in permissioned and even private blockchains. Tendermint's modular design allows Randition's partition formation scheme to be highly flexible and we believe a wide variety of schemes can be implemented with little additional work. Therefore, we suggest a partition formation

scheme that requires only two-thirds majority consensus as a possible improvement over our current one. In Randition's current partition formation scheme, cryptographic sortition is used to randomly and non-interactively determine validators' places in a priority queue to be sorted round-robin into partitions. In this situation, there are no "winners" as there are in Algorand. However, we return to Algorand's usage of cryptographic sortition by suggesting that all validators in a network perform sortition as a lottery to decide which group of validators will *secede* from a Randition sub-network and form a new partition. Informally: once a validator has received sortition results from two-thirds of the network's validators, the set of validators with the highest number of selections by sortition that form the smallest partition size can transition onto Phase 2 and form their own partition; the remaining validators must accept this and also form their own partition. Figure 10 describes this new partition formation scheme.

```
procedure secede(sortitionResults)
   sortByTimesSelected(sortitionResults)
   i ← 0
   min ← minimum partition size
   while i < min do
      validator ← pop(sortitionResults)
      seceders ← validator
      i = i + validator.size
   return seceders
```

Figure 10: The secession scheme

Note that *sortByTimesSelected()* breaks ties on times selected with the higher hash sorting higher. Also note that a minimum partition size can be determined either by a number of validators or by a total voting power.

Again, validators will not have the same chance of being selected given a network with widely varying voting powers. In this scheme, validators with more voting power will

have a higher likelihood of selection and winning the lottery. Supposing a network in which there is a Byzantine participant, it is arguably likely that any winners are leaving behind a partition containing a validator that's misbehaving—unless the Byzantine entity is being more subtle. We leave detailed design and analysis of such a scheme to future work.

Robust query or read support was not implemented in Randition and attempting to query a validator following partitioning will return correct intra-partition data and stale extra-partition data. Currently, one would need to know which validator belongs to which partition to ensure correct data reads. Of course, we intend for user applications to not concern themselves with partitioning and we wish for Randition to fully support reading correct extra-partition data from any validator. Naively, we suggest the following data read scheme: when a validator receives an extra-partition query, it forwards the query to one or few of the appropriate extra-partition validators it knows of and returns the result to the user. Optionally, the validator could forward the request to all extra-partition validators to achieve majority consensus. Of course, this scheme isn't viable for any read-intensive user application. We therefore suggest a major change to Randition whereby each validator observes consensus for all other partitions while maintaining consensus processing on a primary partition. This scheme requires more processing and would likely diminish the write throughput improvement we observe, but ensures that each validator is data consistent. Similar blockchain sharding and partitioning schemes have been proposed in academia. We leave architecture and design for querying or reading data to future work and we could borrow from state-of-the-art research into weak or eventual consistency models.

Randition currently cannot scale downwards, because we have not implemented partition merging. Such a feature would allow users to perform load scaling for applications with volatile workloads. Given a *dissolving* and a *solution* partition in a Randition network, we naively suggest a scheme in which:

1. all validators of the solution partition play back all blocks from the dissolving partition since the initial partitioning event
2. all validators of the dissolving partition reset
3. all validators of the dissolving partition take advantage of Tendermint's Fast Sync to synchronize with the solution partition.

We do not propose each partition play back the other partition's blocks since that would result in differing blockchains across the merged partition. We leave implementation or improvements of this design to future work.

As of the publication of this effort, Tendermint has released version 0.31.5 while Randition is based on Tendermint 0.24.0. We did not routinely pull in updates from Tendermint to avoid the overhead of repeatedly accounting for changed design and behavior. An informal review of Tendermint 0.31.0 logic and discussions since Tendermint 0.24.0 reveal notable improvements in mempool performance in addition to general improvements in consensus, gossip, etcetera. We leave merging the latest Tendermint release into Randition and revalidation of transaction write performance to future work.

Conclusion

To conclude, we show that blockchain partitioning can provide significant write performance improvements at the cost of some Byzantine-fault tolerance. While the performance improvements we observe are likely unsurprising, the key to this paper lies in our partition formation scheme and how Randition can do so safely and with liveness. Conceptually, our ideas can potentially be incorporated into any state machine replication algorithm or model that satisfies the partition transfer protocol requirements. We admit that our partitioning scheme might not be desirable for public, permissioned, or perhaps even private blockchains, but we do believe use cases exist for Randition. We wish to thank the Tendermint and Algorand teams for developing and publishing Tendermint and cryptographic sortition, respectively. We also thank the developers of the partition transfer protocol for their invaluable insight into replicated state machine partitioning.

References

[1] L. Lamport, R. Shostak and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 4, no. 3, pp. 382-401, 1982.

[2] Merriam-Webster, Incorporated, "Blockchain," [Online]. Available: https://www.merriam-webster.com/dictionary/blockchain. [Accessed 3 March 2019].

[3] L. Fortney, "Blockchain, explained," Dotdash, 10 February 2019. [Online]. Available: https://www.investopedia.com/terms/b/blockchain.asp. [Accessed 3 March 2019].

[4] J. Martindale, "What is a blockchain?," Designtechnica Corporation, 11 February 2019. [Online]. Available: https://www.digitaltrends.com/computing/what-is-a-blockchain/. [Accessed 3 March 2019].

[5] Blockchain Luxembourg S.A., "Confirmed transactions per day," [Online]. Available: https://www.blockchain.com/charts/n-transactions. [Accessed 3 March 2019].

[6] BitInfoCharts, "Ethereum transactions historical chart," [Online]. Available: https://bitinfocharts.com/comparison/ethereum-transactions.html. [Accessed 3 March 2019].

[7] E. Buchman, "Tendermint: Byzantine Fault Tolerance in the Age of Blockchains," 2016. [Online]. Available: https://atrium.lib.uoguelph.ca/xmlui/handle/10214/9769. [Accessed 6 May 2018].

[8] Jepsen, "Tendermint 0.10.2," 2017. [Online]. Available: https://jepsen.io/analyses/tendermint-0-10-2. [Accessed 6 May 2018].

[9] Tendermint, "Tendermint core (BFT consensus) in Go," Github, 2018. [Online]. Available: https://github.com/tendermint/tendermint. [Accessed 17 August 2018].

[10] Y. Gilad, R. Hemo, S. Micali, G. Vlachos and N. Zeldovich, "Algorand: Scaling Byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, Shanghai, China, 2017.

[11] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi and K.-L. Tan, "Blockbench: A framework for analyzing private blockchains," in *SIGMOD '17 - Proceedings of the 2017 ACM International Conference on Management of Data*, Chicago, Illinois, US, 2017.

[12] International Business Machines Corporation, "Partition-by-range table spaces," [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSEPEK_12.0.0/admin/src/tpc/db2z_rangepartitionedtablespaces.html. [Accessed 3 March 2019].

[13] Oracle Corporation, "Partitioning concepts," [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/19/vldbg/partition-concepts.html. [Accessed 3 March 2019].

[14] Microsoft Corporation, "Partitioned tables and indexes," [Online]. Available: https://docs.microsoft.com/en-us/sql/relational-databases/partitions/partitioned-tables-and-indexes. [Accessed 3 March 2019].

[15] A. Nogueira, A. Casimiro and A. Bessani, "Elastic state machine replication," *IEEE Transactions on Parallel and Distributed Systems,* vol. 28, no. 9, pp. 2486-2499, 2017.

[16] T. Dasu, Y. Kanza and D. Srivastava, "Unchain your blockchain," in *Symposium on Foundations and Applications of Blockchain (FAB)*, Los Angeles, California, US, 2018.

[17] Tendermint, "Welcome to Tendermint!," Read The Docs, 2018. [Online]. Available: https://tendermint.readthedocs.io/en/master/. [Accessed 17 August 2018].

[18] S. Micali, M. O. Rabin and S. P. Vadhan, "Verifiable random functions," in *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, New York, New York, US, Oct 1999.

[19] E. Buchman, J. Kwon and Z. Milosevic, "The latest gossip on BFT consensus," 2018. [Online]. Available: https://arxiv.org/abs/1807.04938. [Accessed 19 July 2018].

[20] C. Dwork, N. Lynch and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the Association for Computing Machinery (JACM),* vol. 35, no. 2, pp. 288-323, April 1988.

[21] Google, "The Go Project," Google, [Online]. Available: https://golang.org/project/. [Accessed 24 August 2018].

[22] CONIKS Team, "A CONIKS Implementation in Golang," Github, 2018. [Online]. Available: https://github.com/coniks-sys/coniks-go. [Accessed 24 August 2018].

[23] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe and B. Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering,* vol. 2, pp. 77-89, 2012.