Randition: Random Blockchain Partitioning for Write Throughput

David Nguyen  
Department of Computer Science  
San Jose State University  
San Jose, CA  
david.nguyen16@sjsu.edu

Teng-Sheng Moh  
Department of Computer Science  
San Jose State University  
San Jose, CA  
teng.moh@sjsu.edu

Abstract—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.

Keywords—byzantine fault tolerant, state machine replication, blockchain partitioning, elastic partitioning

I. INTRODUCTION

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.

The research community has realized blockchain can address the Byzantine fault weakness in popular consensus algorithms. 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. However, write performance in popular public blockchains are outmatched by virtually all conventional database systems—even distributed databases—available now.

II. 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]. Developers can freely download Tendermint, attach the same deterministic state machine application to each Tendermint validator, and observe their Tendermint network replicate the application. Given at least two-thirds of validators are non-Byzantine and correct, the algorithm can guarantee safety, liveness, and even accountability. When we say a distributed system is safe or maintains safety, we generally mean the system will agree on a single outcome, such as a proposed block commit at a specific blockchain height. When we say a system is live or maintains liveness, we generally mean the system will eventually produce an outcome, such as a proposed block being committed within an expected time frame.

In Tendermint, 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.

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.

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 [12]. They propose a modular partition transfer protocol to allow node groups to split and merge with minimal impact on performance.

III. 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. We believe write throughput maximum is improved if Tendermint takes advantage of the concepts presented in the partition transfer protocol [12].

We propose a Tendermint variant that borrows from this partition transfer protocol, defines how it does partition formation, and specializes in partitionable workloads to provide significant transaction and validator scalability. 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 this variant’s ability to commit as many transactions as possible in a time frame instead of what it commits within a time frame. We call this variant Randition.

We design Randition for users who have highly partitionable workloads within the application they wish to replicate across a network. While blockchains are very synonymous with cryptocurrencies, blockchains also lend themselves to being highly reliable distributed databases, public ledgers, and other similar applications. Tendermint provides a primitive key-value store called kVstore as a basic example application for users.

We understand partitioning 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 Byzantine-fault tolerance is known more as a specific amount rather than as a ratio of total validators. Suppose we have a user who controls 64 validators. Altogether in a single Tendermint network, this network is Byzantine-fault tolerant with up to a maximum of 21 incorrect validators (less than one-thirds of 64). Now, suppose that this user could easily tolerate 5 incorrect validators at any time in any configuration. In this situation, the user could provision a Randition network of 16 validator partitions, because one-thirds of 16 is approximately 5.3. We believe this situation would provide significantly improved transaction throughput at reasonable cost. Suppose a more extreme example in which a user controls a thousand validators. This user might decide they can tolerate far less than 333 validators experiencing Byzantine failures at any time and prefer to gain more performance at the cost of some of this high Byzantine-fault tolerance.

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: High-level architecture of Randition’s partition formation

Figure 1 visualizes 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 safety and liveness for partition transfer by Nogueira et al. shall guide us on the correctness of our partitioning scheme and logic.

IV. ASSUMPTIONS

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.

Our partition formation protocol currently requires 100% consensus, which means every validator must participate in partition formation within a network-wide 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.

V. OVERVIEW

A. Cryptographic Sortition

Verifiable random functions (VRFs) [15] 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 \( \pi \). Micali et al. prove that seed \( x \) is pseudo-random and unpredictable. Anyone who knows \( sk \)'s public key \( pk \) can utilize the verification function \( V \) (consider this an inverse operation of function \( F \)) with input \( pk \), seed \( x \), proof \( \pi \), 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 seeds 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:

\[
<\text{seed}_r, \pi> \leftarrow \text{VRF}_{\text{sk}}(\text{seed}_r || r)
\]  

(1)

where \( \pi \) is proof of seed\(_r\)'s validity and \( \text{VRF}_{\text{sk}} \) represents the execution of the VRF with user \( u \)'s secret key.

Of course, proposer \( u \)'s public key is known by every other user. Proposer \( u \) includes the output seed\(_r\) and proof \( \pi \) in its block proposal. As part of block validation, all other users validate \( u \)'s seed, with \( \pi \). If the block is approved for commit, the block is appended to the blockchain along with seed.

Since the VRF 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: The cryptographic sortition algorithm [10].](image)

Figure 2 summarizes the sortition algorithm [10]. Users input a private key \( sk \), a pseudo-random public \( seed \), the desired number of users \( \tau \), desired \( role \), the user’s weight \( w \), and the network's total weight \( W \). A pseudo-random \( hash \) is produced from the blockchain’s public seed concatenated with the user's desired role, such as a proposer or voting committee identifier. A pseudo-random number in the interval \([0, 1)\) is calculated by dividing the hash by the maximum numerical value of the hash. Probability \( p \) is defined as the desired number of users \( \tau \) 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. Counter \( j \) indicates how much of the user's weight was selected by the algorithm or "won" sortition.

The interval \([0, 1)\) is also divided into 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 low \( j \) with high probability and vice versa. The algorithm finally outputs the number of selections \( j \), the pseudo-random \( hash \) it used to determine \( j \), and proof \( \pi \) of the validity of hash altogether in a sortition message.

![Figure 3: The cryptographic sortition verification algorithm [10].](image)

Figure 3 summarizes the sortition verification algorithm [10]. If a node is selected for a role and publicly broadcasts the results of their sortition, anyone can verify the results with this algorithm along with the originating node's public key \( pk \). Cumulatively, the sortition and verification algorithms are essential in providing safe and deterministic selection of committees.

B. Partition Transfer

Nogueira et al. iterate that modern state machine replication primarily provides fault tolerance at the cost of scalability [12] 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 the following 6 steps [12]:

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 \( \Delta \).
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 \( \Delta \) 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 \( \Delta \) when it receives and verifies matching \( \Delta \)-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.

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.

C. Tendermint

While Tendermint’s core design remains the same, observation of Tendermint’s code repository [9] and public documentation [14] over time indicate the entire product is in a constant state of improvement and refactoring. Modifying a specific version of Tendermint alleviated problems associated with alpha and beta software updates. As such, we only worked with and discuss Tendermint version 0.24.0.

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 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. Tendermint’s Validator Set Update protocol [7] [14] is integral to our effort. It is triggered when the user application specifies a new validator set at the EndBlock request.

VI. ADAPTED CRYPTOGRAPHIC SORTITION

We modify Tendermint’s consensus reactor and engine to maintain a pseudo-random seed value at every height in the blockchain. 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 broadcast 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 tracking the validator’s partition and partition status, tracking the network’s partitions and their status, and 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.

Figure 4 illustrates the valid transitions between the phases.

![Figure 4: Partition reactor states](image)

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 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 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 a part 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. 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 to partitions in round-robin order. Note that formation into only 2 partitions is supported at this time, although the scheme could easily be improved to support a dynamic number 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.

VII. ADAPTED PARTITION FORMATION

Randition is very much inspired by Nogueira et al.’s proposal and we don’t precisely implement the defined protocol. Instead, we rely on it to provide valuable guidance and insight towards maintaining safety and liveness during blockchain partitioning.

Consider the following Randition’s partition formation protocol, inspired by Nogueira et al.’s partition transfer protocol:

I. Group G receives a partition transfer request from a trusted agent.

II. Group G collectively and consistently determines new groups L1 and L2, known as partition formation.

III. Groups L1 and L2 cease participating in consensus with each other and continue consensus locally.

IV. Groups L1 and L2 report the partition transfer request succeeded to the trusted agent.

Nogueira et al. [12] state their 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” [12]
- **Safety 2**: “linearizability of the service is preserved by the partition transfer” [12]
- **Liveness**: partitioning eventually completes

Our partition formation protocol satisfies these goals in an equivalent manner. Randition satisfies Safety 1 due to Steps II and III. Partition formation in Step II ensures two or more independent partitions. At the very least, all validators in a new partition agree they’re in a partition with each other. At best, all validators in a whole network agree on which partitions all other validators belong to. Step III ensures independent “group states” and data consistency, because every partition does not process extra-partition requests. Every validator in a partition will not propose or commit transactions that are rejected by their partition’s partitioning key. Safety 2 is satisfied if the protocol processes requests in total order. 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 arrive and they are all valid, partition formation occurs and completes independently. If they do not arrive, then Randition carries on and another partition request and sortition message gossip must be made in a future round. Steps III and IV execute independently of other intra-partition validators and rely on existing Tendermint behavior. 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. With our partition formation protocol and existing Tendermint behavior, Randition satisfies the safety and liveness goals outlined by Nogueira et al.

Now consider Figure 4, where each partition phase is represented as a state in a state machine. This begs the question regarding if state transitions within the partition reactor are safe and live. Can a correct validator be coerced into incorrect states or forced to not terminate? Within Randition, only the partition reactor is 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.

The primary 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 can no longer be considered 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 guaranteed to execute correctly as long as Tendermint can continue processing rounds.

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 sortition and gossip since requiring 100% consensus within one or two rounds is admittedly a lofty expectation. Other than successfully transitioning to Phase 2 via 100% consensus, the only other way to guarantee termination of Phase 1 is for the user application to explicitly request cancellation. 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. The partition reactor does not block core processing in this regard. Phase 2 terminates because partition formation is completed independently of other validators. 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.

Note that Randition does not take additional action for extra-partition data that was already committed before partitioning. Therefore, a validator in a partitioned blockchain will eventually have stale data. 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.

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.

VIII. IMPLEMENTATION

Tendermint’s source code is written in Golang (also known as Go), an open source programming language whose development is led by Google [18]. 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 [19]. Both Tendermint and Coniks use Ed25519 [20] 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.

IX. 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. 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.

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.

We are mainly interested in observing transaction throughput after the blockchain partitions itself. Therefore, each individual experiment is conducted after we prime the test Randition network by submitting transactions for 20 seconds, which is enough time to allow Randition to partition itself and settle network activity. 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 transaction rate per validator between 25, 50, 75, 100, 150, and 200 transactions per second. That translates to 800, 1600, 2400, 3200, 4800, and 6400 input transactions per second over a whole 32-validator network. Each experiment executes for 20 seconds.

![Figure 5: Committed transactions per second](image)

![Figure 6: Committed blocks per second](image)

Figure 5 compares transactions committed per second between Tendermint and Randition. Note that the error bars indicate the standard deviation. Figure 6 compares blocks committed per second between Tendermint and Randition. Note that the error bars indicate the standard deviation. These results are comprised of 12 executions per input rate for Tendermint and 16 executions per input rate for Randition.

X. DISCUSSION

To reiterate, this effort focuses on proving that transaction write performance is significantly improved when safely partitioning a blockchain. In Figure 5, 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. Randition reaches its peak committed-transactions-per-second past Tendermint’s peak and continues to maintain better performance than that of Tendermint. In Figure 6, 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.

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. 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. Naturally, consensus requirements will tend to increase non-linearly with more participating nodes. The computations required to process proposals, blocks, and votes; and the time required for proposal, block, and vote messages to propagate sufficiently to achieve consensus are major contributors to these requirements.

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 if there are 2 partitions. One might consider this a natural optimization of blockchain partitioning, but only if they 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.

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 Section VII that Byzantine validators cannot coerce the network into performing incorrect partition formation or trick a validator into joining the wrong or multiple partitions.

XI. FUTURE WORK

Randition requires much improvement and iteration to approach becoming a useful application. This particular effort has focused on developing a prototype to demonstrate feasibility.
We concede that 100% consensus is an unrealistic expectation in permissioned and perhaps 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 sorting 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 sorting 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 might transition onto Phase 2 from Figure 4 and form their own partition; the remaining validators must accept this and also form their own partition. Figure 7 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 7: The secession scheme

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 sortition. We leave detailed design and analysis of such a scheme to future work.

**XII. CONCLUSION**

To conclude, we show that blockchain partitioning can provide significant write performance improvements at the cost of some BFT. 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 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/confirmed-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://arxiv.org/abs/1807.04938. [Accessed 19 July 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] 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.
[13] T. Dasu, Y. Kanza and D. Srivastava, "Unchain your blockchain," in Symposium on Foundations and Applications of Blockchain (FAB), Los Angeles, California, US, 2018.
[14] Tendermint, "Welcome to Tendermint!", Read The Docs, 2018. [Online]. Available: https://tendermint.readthedocs.io/en/master/. [Accessed 17 August 2018].
[15] 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.
[16] 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].
[17] 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.
[18] Google, "The Go Project," Google. [Online]. Available: https://golang.org/project/. [Accessed 24 August 2018].
[19] CONIKS Team, "A CONIKS Implementation in Golang," Github, 2018. [Online]. Available: https://github.com/coniks-sys/coniks-go. [Accessed 24 August 2018].
[20] 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.