Gosig: Scalable Byzantine Consensus on Adversarial Wide Area Network for Blockchains

Article · February 2018

CITATIONS
0

READS
129

4 authors, including:

Peilun Li
Tsinghua University
3 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Guosai Wang
Tsinghua University
4 PUBLICATIONS **14** CITATIONS
SEE PROFILE

Gosig: Scalable Byzantine Consensus on Adversarial Wide Area Network for Blockchains

Peilun Li
Tsinghua University

Guosai Wang
Tsinghua University

Xiaoqi Chen
Princeton University

Wei Xu
Tsinghua University

Abstract

Existing Byzantine fault tolerance (BFT) protocols face significant challenges in the consortium blockchain scenario. On the one hand, we can make little assumptions about the reliability and security of the underlying Internet. On the other hand, the applications on consortium blockchains demand a system as scalable as the Bitcoin but providing much higher performance, as well as provable safety. We present a new BFT protocol, Gosig, that combines crypto-based secret leader selection and multi-round voting in the protocol layer with implementation layer optimizations such as gossip-based message propagation. In particular, Gosig guarantees safety even in a network fully controlled by adversaries, while providing provable liveness with easy-to-achieve network connectivity assumption. On a wide area testbed consisting of 140 Amazon EC2 servers spanning 14 cities on five continents, we show that Gosig can achieve over 4,000 transactions per second with less than 1 minute transaction confirmation time.

1 Introduction

The rise of cryptocurrencies, such as Bitcoin [43], increases the awareness and adoption of the underlying *blockchain* technology. Blockchain offers a distributed *ledger* that serializes and records the transactions. Blockchain provides attractive properties such as full decentralization, offline-verifiability, and most importantly, scalable Byzantine fault tolerance on the Internet. Thus, Blockchain has become popular beyond cryptocurrencies, expanding into different areas such as payment services, logistics, healthcare, and Internet-of-Things (IoT) [8, 54, 48].

While Bitcoin provides a *permissionless* protocol, where everyone can join, we focus on *consortium blockchains* (aka *permissioned* blockchains), where a participant needs offline authentication to join. It is useful in many commercial applications [25]. While we no longer have to worry about Sybil attacks [21], there are still some other significant challenges.

The blockchain is replicated to each participant, and the key problem is how to reach consensus on these replicas. Comparing to traditional distributed transaction systems, the three biggest challenges of a blockchain are:

1) Players are from different organizations without mutual trust. Failures, even Byzantine failures, are common. Thus, we cannot rely on a small quorum (e.g., Chubby [9] or ZooKeeper [27]) for consensus. Instead, we need Byzantine consensus and allow all players to participate, i.e., supporting a consensus group with thousands of servers.

2) The system runs on an open Internet. The network can drop/delay communication arbitrarily. Even worse, as the addresses of the participants are known to all, an adversary can easily launch attacks targeting any chosen participant at any time, using techniques like distributed denial-of-service (DDoS) [26, 49]. The adversary can strategically choose which players to attack, and victims will remain unavailable until the adversary *adaptively* choose to attack others. We call these attacks *adaptive attacks*, which strongly threatens the special nodes in a protocol, such as the leaders.

3) Different from Bitcoin that allows temporary inconsistency (i.e., a *fork*), people usually expect a permissioned chain to provide more traditional transaction semantics, i.e., a committed transaction is durable. Also, applications on a permissioned chain [1] require much higher throughput and lower latency than Bitcoin.

While there are many Byzantine fault-tolerant (BFT) protocols, most of them do not sufficiently address these three challenges. For example, PBFT [13] and its successors [33, 38] and even some Bitcoin variants [23] all depend on a leader or a small quorum to participate multiple rounds of communications, and thus they are vulnerable to *adaptive attacks* on the leader. Other protocols that try to avoid faulty leaders by changing leaders in a fixed order [35, 42, 51, 3] cannot avoid adaptive leader

attacks, because once the adversaries know who is the next leader, they can change their target accordingly.

There is a new generation of BFT protocols designed to run over the Internet. To avoid adaptive attacks, Algorand [24] hides the leader identity. To improve scalability, ByzCoin[32] combines Proof of Work (PoW) with multi-signature-based PBFT. To tolerate arbitrary network failures, HoneyBadgerBFT[41] adopts asynchronous atomic broadcast [11] and asynchronous common subset (ACS) [4].

Unfortunately, as we will detail in Section 3.3, none of these BFT protocols offer the following properties at the same time: 1) liveness under *adaptive attack*, 2) scalability to 10,000s of nodes with low latency (15 seconds in our simulation) for commitment, and 3) provable safety (i.e., no fork at any time) with arbitrary network failure. Also, there is no straightforward way to combine the techniques used in these protocols.

We present Gosig[1], a new BFT protocol for permissioned blockchains. Gosig can achieve all three properties above, and also provide provable liveness with partially synchronous network (details in Section 3.2).

Gosig elects different leaders secretly for every block, and it eliminates the leader's involvement after it proposes a block to defend against adaptive attacks on leaders. At the implementation level, we use gossip-based communications to fully exploit the link redundancy on the Internet while keeping the source safe. Since we need to gather signatures during gossip, we adopt asynchronous multi-signature [6, 46] to largely reduce the network overhead of consensus messages.

We evaluate Gosig on an Amazon EC2-based 140-server testbed that spans 14 cities on five continents. We can achieve a throughput of 4,000 tps (transactions per second) with an average transaction confirmation latency of 1 minute. Even when 1/4 of the nodes fail, we can still maintain over 3,500 tps with the same latency. With over 100 participants, it effectively doubles the throughput and reduces the latency by 80% comparing to HoneyBadgerBFT [41], the state-of-the-art protocol that offers the same level of safety guarantee. Also, using simulations, we show that Gosig is able to scale to 10K nodes.

In summary, our major contributions are:

1) We propose a new BFT protocol that achieves scalability, provable safety and resilient to adaptive attack.

2) We propose a novel method of combining *secret leader selection, random gossip, multi-round voting*, and *multi-signature* into a single BFT protocol in a compatible way.

3) We provide a real Gosig implementation, evaluate it on a real-world geographically distributed testbed with 140 nodes, and achieve promising performance results.[2]

[1]The name is a combination of Gossip and Signature aggregation, two techniques we use.

2 Related Work

Bitcoin and its variants. Permissionless public blockchains like Bitcoin [43], Ethereum [53], PPCoin [31] need proof of work (PoW) or proof of stake (PoS) to prevent Sybil attacks. They also need incentive mechanisms to encourage people to join the public network to keep the system safe. Other designs [17, 20] try to avoid chain forking but retain the design of PoW or PoS. We assume consortium blockchains [10, 45, 52], and mainly focus on the performance and safety of the system, instead of the other economic aspects.

Byzantine fault tolerance. The most important feature of a BFT protocol is safety. Unfortunately, many open source BFT protocols are not safe [12]. There are two major approaches to design provable BFT agreement protocols. 1) Using multi-round voting: example systems include PBFT [13] and its successors [33, 38, 14]; 2) Using leader-less atomic broadcast: HoneyBadgerBFT [41] and [16, 34]. To prevent malicious leaders from affecting the system, Aardvark [15] use performance metrics to trigger view changes and Spinning [51], Tendermint [35] or others [3, 42] rotates leader roles in a round robin manner. However, there methods can not avoid adaptive attacks because the leader role is known to all in advance, and thus can be muted by attacks like DDoS right before it becomes a leader. Gosig adopts similar voting mechanism like PBFT to get good performance without failure, and keeps safety and liveness under attacks.

In order to scale the system, many systems adopt the "hybrid consensus" design [30, 32, 44] that uses a Bitcoin-like protocol to select a small quorum, and use another (hopefully faster) BFT protocol to commit transactions. If adversaries can instantly launch *adaptive attacks* on leaders, it is hard for these protocols to maintain liveness. Algorand [24] leverages secret leader election and quorum member replacement methods to keep liveness. Gosig lets every player participate in the consensus, but combines similar secret leader selection with signature-based voting to prevent such attacks.

We use similar methods and adversary models proposed in Algorand [24]. We adopt the idea of multi-round voting from PBFT and HoneyBadgerBFT [41], and the idea of multi-signature from ByzCoin [32]. Gosig combines these incompatible methods in a coherent protocol and achieves good performance. We compare the key differences of these protocols in Section 3.3.

Overlay network and gossip. Most BFT protocols and blockchains use broadcast as a communication primitive.

[2]We will opensouce Gosig when the paper is published.

To improve broadcast reliability on the Internet, people often use application-layer overlay networks. We adopt techniques like gossip from reliable multicast [5], probabilistic broadcast [22, 29] and other peer-to-peer (P2P) networks [28, 50]. Existing P2P networks may tolerate some Byzantine failures, but do not provide convergence guarantee [37]. By combining network optimizations like gossip with a robust protocol layer design, we can greatly improve both system resilience and availability.

3 Problem Definition and Assumptions

The goal of Gosig is to maintain a blockchain. In Gosig, clients submit *transactions* to *players* (or servers), who pack these transactions into *blocks* in a specific order. All committed blocks are then serialized as a *blockchain*, which is is replicated to all players. On the blockchain, one block extends another by including a hash of the previous block. In a blockchain, a transaction is *confirmed* only when the consensus group *commits* the block containing the transaction. Gosig, as a consensus protocol, ensures that all blockchain replicas are the same. In particular, we want to prevent *forks* on the blockchain, i.e. two different blocks extending the same block.

3.1 Problem Definition

We consider a system with N players, $p_1, p_2, \cdots, p_n$. We can tolerate b static Byzantine failures and c *adaptive attacks* where $b + c = f = \lfloor (N-1)/3 \rfloor$. The Byzantine faulty nodes can do anything including colluding to break the protocol. The honest players under *adaptive attacks* act like crash failure, but they come to life again when the attacks are over. All other (at least $2f+1$) players are *honest* and follow the protocol.

All players form a *consensus group* $\mathbf{G}$. Each player p_i outputs (commits) a series of ordered blocks $B_i[1], B_i[2], \cdots, B_i[n_i]$, where n_i, the length of the blockchain after attaching a new block, is the *height* of this block.

A transaction is an operation on a state machine, and we say it is valid when it is a legal operation on the current state. A block $B_i[h]$ is valid if: 1) all transaction included are valid if executed sequentially, and 2) the block header contains the correct reference to the previous block $B_i[h-1]$, like the Nakamoto blockchain [43].

The goal of Gosig is to let $\mathbf{G}$ reach a *consensus on the blockchain*, i.e. the following two conditions hold.

1. Safety: (1) Any block committed by any honest player is valid; (2) at any time, for any two honest players p_i and p_j, $B_i[k] = B_j[k]$ for any $k \leq \min(n_i, n_j)$.

2. Liveness: For any finite time t and any honest player p_i, there exists a time $t' > t$ when p_i commits a block packed by an honest player.

Here we define safety and liveness of blocks instead of transactions for simplicity. We rely on gossip mechanisms to ensure that a transaction will reach most players, and an honest player will pack the transaction when it becomes the leader. Intuitively, the safety condition requires a total order of committed blocks on the blockchains at all honest players, meaning there is no fork at any time. The liveness condition says that all honest players will always make progress, i.e., if a transaction can reach all honest players, it will eventually be confirmed. Both conditions are based on certain assumptions about the system, and we detail them next.

3.2 System Model and Assumptions

We summarize our key assumptions that we use to prove the safety and liveness of Gosig.

Strong cryptography and PKI. We only consider permissioned chain, and there is a trusted public key infrastructure (PKI) to authenticate each player - a common assumption in today's Internet. We also assume the correctness of the cryptographic primitives. These assumptions are the foundation of the safety in Gosig.

Asynchronous network for safety. Our protocol can keep safety under *asynchronous network* [41] condition, which means messages can be arbitrarily dropped, reordered, or duplicated.

Liveness under partial synchrony. We also guarantee *liveness* if the network has *partial synchrony*, a common assumption [14, 33, 42, 51]. We say a network has partial synchrony if there exists a time t' such that for any time $t > t'$, all messages sent between any two honest players during the interval can be delivered within a limited time bound Δ_t. Specially, we assume *adaptive attacks* on any player only take effect after a delay of Δ_t at any time.

Partially synchronized clock. Similar to [24], we assume a partially synchronized clock for getting liveness. That is, at any wall clock time t, for any two players p_i and p_j, their local time readings $C_{p_i}(t)$ and $C_{p_j}(t)$ satisfy that $|C_{p_i}(t) - C_{p_j}(t)| < \Delta$. Practically, it is easy to ensure a Δ of several seconds using standard Network Time Protocol (NTP) on the Internet.

3.3 Key Features of Gosig

Comparing to existing blockchains and other Byzantine-fault-tolerant atomic ordered broadcast protocols, Gosig has achieved *scalability*, *liveness under adaptive attacks* and *safety under asynchronous networks* at the same time. While existing protocols provide one or more of these features, to our knowledge, Gosig is the first protocol that offers all three together.

ByzCoin [32] offers excellent scalability by combining PBFT, signature collection and proof-of-work (PoW). However, like PBFT, it loses liveness under adaptive attacks given that it is still PBFT-based. Even without PBFT, its two-phase multi-signature and Bitcoin-NG [23]-like mechanism that allows elected leaders to keep serving are also vulnerable to adaptive attacks.

Algorand [24] has excellent scalability, and tolerate adaptive attack using secret consortium election. However, the safety of its Byzantine Agreement is based on a *weak synchrony* assumption about the network. This additional requirement comes from the idea of randomly selecting a small quorum, which is the key to Algorand's scalability. We only adopt the secret *leader* election from Algorand to avoid adaptive attacks, but completely redesign the BFT protocol using multi-round signature collections to achieve provable safety in asynchronous networks, like PBFT. We solve the scalability problem by combing protocol design with implementation optimizations like multi-signatures.

HoneyBadgerBFT [41] achieves provable optimal liveness and provable safety in any situation. However, each node needs to send $O(N^2)$ messages per round. Batching up $O(N^2 \log N)$ transactions per round helps amortize the cost, but the large batch results in a latency as high as $O(N^2 \log N)$, limiting the scalability. In comparison, the network overhead for each Gosig player is $O(N \log N)$ per round. Therefore, experimentally, we can double the HoneyBadgerBFT throughput with only 1/5 of the latency on a similar testbed with more participates.

In summary, we insist that Gosig has provable safety under a strong adversary model, but we choose to relax the liveness goal a little, in exchange for better scalability. We achieve this goal by adopting some originally incompatible ideas and provide alternative implementations, so they can be combined seamlessly with our accordingly designed protocol.

4 Gosig Protocol Overview

We provide an intuitive overview of Gosig and leave formal descriptions and analysis to Section 5.

Players. Every players participates in the protocol, and knows all other players' public key. It receives transactions submitted by clients and gossip transactions among all players. An honest player is responsible for verifying transaction validity and block validity. A player drops invalid transactions and blocks, and blacklist the senders.

Rounds and stages. Gosig is a partially-synchronous protocol. We divide the execution of Gosig into *rounds* with a fixed time length (30 seconds by default). Each round consists of a leader selection step (no communications) and two subsequent *stages* with fixed length. Thus, all players know the current round number and stage by referring to the local clock.

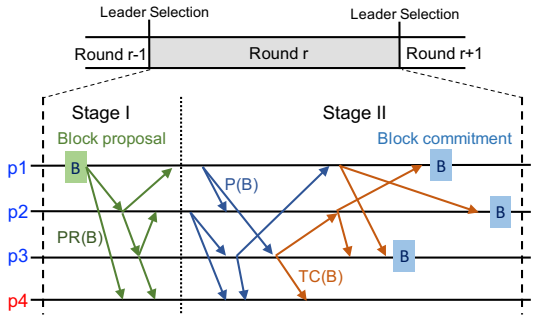


Figure 1: Overview of a Gosig round (happy path only).

Figure 1 provides an overview of a typical round. At the start of each round, some players secretly realize they are *potential leaders* of this round with the cryptographic sortition algorithm (Section 5.2). Thus, the adversary can not target the leaders except randomly guessing.

At Stage I, a selected potential leader packs non-conflict uncommitted transactions into a block proposal, disseminates it with gossip, and acts just like a normal node afterwards. Note that a potential leader's identity can only be discovered by others (including the adversary) after she has full-filled her leader duties.

The goal of Stage II is to reach an agreement on some block proposal of this round by vote exchange. A player "votes" for a block by adding her digital signature of the block to a prepare message ("P message") and sending it. An honest player only votes for a single proposal per round. Upon receiving at least $2f+1$ signatures from P messages for a block, the player starts sending tentatively-commit messages ("TC message") for it. She finally commits the block to her local blockchain replica once she receives $2f+1$ TC messages.

The above process only covers the "happy path" of the protocol. We provide the details how Gosig handles failures in the next section.

5 Gosig Protocol Details

We formally describe the Gosig protocol details. Throughout the paper, we adopt the following notation: subscripts denote who have signed the message[3] and superscripts denote the round in which the message is sent. For example, P_X^r is a P message collectively signed by the set X of players in round r[4]. We also use a shorthand to describe the action that p_i sends a P message about a

[3] A message may be collectively signed by multiple players.
[4] For brevity, we denote $M_{\{i\}}^r$ by M_i^r.

block B as "p_i Ps B", and the action that p_i sends a TC message about a block B as "p_i TCs B".

5.1 Player's Local State

We describe each player as a finite state machine in Gosig. Each player maintains a local state and decides her next actions based on the state and external events (receiving a certain message or selected as a leader).

The local state on player p_i is a 4-tuple $s_i = \langle B_{root}, h_{root}, B_{tc}, F \rangle$. The tuple contains two types of information. The first two values are about the last committed block in her local blockchain copy. B_{root} is the block itself, and h_{root} is the *height* of B_{root}.

The rest values in s_i describe the *pending block* that the player plans to add to her chain at height $h_{root}+1$. B_{tc} is the pending block itself. B_{tc} is non-null if p_i has TCed B_{tc}. Otherwise, B_{tc} is a null value ε. The last variable F characterizes the timeliness of the block of B_{tc}.

Besides the elements appearing in the tuple, p_i also implicitly maintains two variables: $\overline{c_{root}}$ and c_{tc}. $\overline{c_{root}}$ is a *commitment certificate* that proves the validity of B_{root}, and c_{tc} is a *proposal certificate* of B_{tc}. We define both later in this section. For brevity, we do not explicitly include $\overline{c_{root}}$ and c_{tc} in the s_i tuple.

When Gosig starts running, a player's local state is initialized as $s_i = \langle B_\varepsilon, 0, \varepsilon, 0 \rangle$.

5.2 Leader Selection: The First Step

Leader selection is the first step of each round. The objective of this step is to secretly and randomly select the *potential leaders* who are entitled to propose a next block. The greatest challenge is to keep the election result unpredictable until the potential leaders have sent out the propose messages (*PR* message). Otherwise, the adversary can attack the leaders beforehand, and thus break the liveness of the system.

The cryptographic sortition algorithm. We use a simplified version of the *cryptographic sortition* mechanism from Algorand [24] to select a set of *potential leaders*. Similarly, we use a number Q^h to implement cryptographic sortition. In Gosig, Q^h is recursively defined as

$$Q^h = H(SIG_{l^h}(Q^{h-1})) \quad (h > 0) \tag{1}$$

where h is the height of a committed block B[5], H is a secure hash function shared by all players, l^h is defined as the leader who has proposed the block B (the signer of the proposal certificate of B), and $SIG_i(M)$ is the digital signature of message M signed with p_i's private key.

Based on Q^h, we define a player p_i's *leader score* $L^r(i)$ at round r as $L^r(i) = H(SIG_i(r, Q^h))$, where h is the height of the latest committed block at round r.

[5] Q^0 is a random number shared among all players.

At the beginning of each round r, each player p_i computes her $L^r(i)$, and if the score is less than a *leader probability parameter* q, she knows that she is a *potential leader* of the round. A potential leader can prove to other players about her leader status with the value $SIG_i(r, Q^h)$. We define the value as the *leader proof* for round r, lc_i^r. The process requires no communication among players.

The cryptographic sortition algorithm has two good properties: 1) the signature SIG_i uses p_i's private key, and thus cannot be forged by others; 2) If the hash function $H(\cdot)$ is perfectly random, the potential leader selection is uniformly random. Thus, there is no way for the adversary to know who is selected, nor can it change any node's chance of becoming a leader.

Choosing a right q is important. If q is too small, some rounds may not have a leader and thus fail to proceed. If q is too large, there may be many competing potential leaders, wasting resources to resolve the conflict. Similar to Algorand, we set $q = 7/N$ where N is the total number of players. This q is sufficiently large to reduce the probability of no-leader rounds to less than 0.1%.

Of course, a faulty node may still become a potential leader. In this case, the worst damage it can cause is to stall the round without affecting the correctness.

5.3 Potential Leaders Propose Blocks

When a player p_i recognizes that she is a potential leader, she needs to decide which block to propose and then generate a proposal message.

Given p_i's current local state $s_i = \langle B_{root}, h_{root}, B_{tc}, F \rangle$ (B_{tc} may be ε), she first gathers a list of candidate blocks to propose. If p_i finds she has a non-empty B_{tc}, the block B_{tc} is a good candidate to propose again, and the height stays the same as B_{tc}. Alternatively, she can also construct a new block extending her own chain, at height $(h_{root}+1)$. The following procedure defines which block she will propose.

To make a proposal valid, a potential leader needs to provide a valid *certificate* c for the proposed block B at height h. In addition to serving as a proof of the block validity, c also determines the *proposal round* $r_p(B)$[6] of the block proposal. The leader decides which candidate block to propose based on $r_p(B)$.

We can understand the proposal round as the round number when the block is first generated. Intuitively, a block with a larger proposal round is more likely to get accepted by peers. Therefore, in Gosig, we stipulate that a potential leader should propose the block with *the largest proposal round* among all candidate blocks.

[6] More precisely, the proposal round is an attribute of a proposal, which is defined in the next paragraph, rather than of a block. We use $r_p(B)$ notation for brevity.

A proposal certificate c is *valid* if and only if it matches one of the following two cases. We also specify how we compute the proposal round in each case.

Case 1: $c = TC_i^{r'}(B^{r'}, h)$ $(r' < r)$, where B and h are exactly the proposed block and its height, respectively. In this case, $r_p(B) = r'$;

Case 2: $c = TC_X^{r'}(B^{r'}, h-1)$ $(r' < r)$, where X contains at least $2f+1$ different players. In this case, $r_p(B) = r'+1$.

Finally, the potential leader p_i assembles a proposal message (*PR* message) in the form of $PR_i^r(B^*, h, c, lc)$ containing: 1) the proposed block B^*, 2) B^*'s height h, 3) the proposal certificate c, and 4) the leader proof lc (defined in Section 5.2). Then p_i signs the message with her private key. Everyone can easily verify the validity of the *PR* message by checking the included block, signatures and certificates.

5.4 Stage I: Block Proposal Broadcast

After the leader selection step, Gosig enters Stage I: block proposal dissemination. The objective of this stage is to propagate blocks proposed by all potential leaders to as many honest players as possible.

We use the well-known gossip protocol [47] to disseminate messages on the application layer overlay network formed by the same set of players. A player sends/forwards a message to m randomly selected players in each hop. Parameter m is called the *fanout* and determines how fast the message propagates.

Potential leaders initiate the gossip of their *PR* messages. Each honest player, upon receiving a *PR* message, will first check its validity, and then forward all valid *PR* messages. Note that there can be more than one valid block proposed in the same round, either because there are multiple potential leaders, or because a malicious leader can propose multiple valid blocks. Players forward all valid blocks in this stage and leave the conflict resolution to Stage II.

At the end of Stage I (after a fixed time interval T_1), we expect that most players have seen all block proposals in the round, assuming everything goes well. Nevertheless, Stage II is able to handle all complicated situations.

5.5 Stage II: Signature Collection

The objective of Stage II is to disseminate signed messages of players' votes for the block proposals, in the hope that honest players can commit a single valid block. Same as Stage I, we use gossip to propagate all messages.

Message types. There are two message types involved in this stage. Players can collectively sign a message by appending their signatures. We say a message M_X^r is *k-signed* if X contains *at least* k distinct players' signatures.

***P* message.** A prepare message (*P* message) digitally signs a block. Specifically, $P_i^r(B^r, h)$ means player p_i signs her vote for the block B^r at height h proposed in round r. In short, we say p_i prepares or "*P*"s B^r.

***TC* message.** A tentatively-commit message (*TC* message) signs a proof of a $(2f+1)$-signed *P* message. Specifically, $TC_i^r(B^r, h)$ proves that at least $2f+1$ players (including p_i) have *P*ed the block B^r at height h in round r. In short, we say p_i tentatively-commits or "*TC*"s B^r.

Stage II protocol. Algorithm 1 outlines the expected behavior of an honest player p_i in Stage II. We model each p_i as a finite state machine with local states listed at the beginning of Algorithm 1. It performs actions based on current local state and the incoming messages.

Lines 1 to 7 describe the initialization procedure, in which p_i checks all block proposals she receives in Stage I (by calling the function *DecideMsg* in Algorithm 2). If p_i received valid proposals in Stage I, she needs to decide which block to prepare (function *DecidePMsg* in Algorithm 2). In general, p_i prefers a block proposal with larger proposal round, as it indicates a more recent block (lines 2.14 to 2.20). Finally, p_i chooses exactly one block $\bar{B}$ for height h, and *P*s it (line 1.4 and line 1.7, respectively).

After initialization, the state machine of p_i starts to handle incoming messages. Lines 8 to 29 in Algorithm 1 outlines handler routines for these three different message types. p_i only signs messages about the same block that she has *P*ed (line 1.9 and line 1.21), she can only *TC* a block B after she collects at least $2f+1$ signatures from the *P* message about B, and she can only commit a block B after she collects at least $2f+1$ signatures from the *TC* messages about B (line 1.13 and 1.25). These rules ensure the safety of Gosig.

5.6 Reducing Signature Sizes

Gosig protocol requires signatures from over 2/3 of the players. To reduce the storage and communication overhead of signatures, we adopt the techniques in [7, 6] to aggregate these signatures into a compact *multi-signature* form.

The cryptographic signature of a player p_i involves a hash function H, a generator G, a private key x_i, and a public key $V_i = G^{x_i}$. A player holding the private key x_i can sign a message M by computing $S_i = H(M)^{x_i}$, and

[7] Other players' signatures are in the received messages.

[8] Among all blocks with the largest proposal round in S, $\overline{B}$ is the block whose proposer has the smallest leader score.

[9] Note that $r_p(B')$ is actually the proposal round of the proposal message about B', which is not necessarily equal to $r_p(B_{tc})$.

Algorithm 1 Stage II workflow for each player p_i.

Constants:
– **G**: the consensus group of N players
– r: the current round number
State Variables:
– s_i: p_i's local state, i.e., $\langle B_{root}, h_{root}, B_{tc}, F\rangle$
– S: the set of all valid proposals received in Stage I

$phase \leftarrow$ Init
$msg \leftarrow$ DECIDEMSG ▷ See Algorithm 2
if $msg \neq null$ **then**
  $P(\overline{B}, h) \leftarrow msg$ ▷ $\overline{B}$ is the block p_i votes for
  $phase \leftarrow$ Ped
  $X_P \leftarrow \{i\}$ ▷ The players that have *P*ed $\overline{B}$
  Prepare $\overline{B}$ by gossiping msg

On receiving a valid $P^r_{X'}(B,h)$ message **Do**
  if $phase =$ Ped **and** $B = \overline{B}$ **then**
    $X_P \leftarrow X_P \cup X'$
    $sig_P \leftarrow$ signatures of players in X_P[7]
    Sign $P^r_{X_P}(B,h)$ with sig_P
    if $P^r_{X_P}$ is (2f+1)-signed **then**
      $phase \leftarrow$ TCed
      $X_{TC} \leftarrow \{i\}$ ▷ The players that have *TC*ed $\overline{B}$
      $s_i \leftarrow \langle B_{root}, h_{root}, B, r\rangle$
      Tentatively commit B by gossiping $TC^r_i(B,h)$
    else
      Forward the signed message $P^r_{X_P}$ with gossip

On receiving a valid $TC^r_{X'}(B,h)$ message **Do**
  if $phase \neq$ Init **and** $B = \overline{B}$ **then**
    $X_{TC} \leftarrow X_{TC} \cup X'$
    $sig_{TC} \leftarrow$ signatures of players in X_{TC}
    Sign $TC^r_{X_{TC}}(B,h)$ with sig_{TC}
    if $phase \neq$ Ced **and** $TC^r_{X_{TC}}$ is (2f+1)-signed **then**
      $s_i \leftarrow \langle B, h, \varepsilon, 0\rangle$
      Commit B on the local blockchain
      $phase \leftarrow$ Ced
    Forward the signed message $TC^r_{X_{TC}}$ with gossip

Algorithm 2 Deciding which block to prepare in Stage II.

function DECIDEMSG
  if $S = \phi$ **then** ▷ Received no valid proposals
    return null
  else ▷ Received one or more valid proposals
    return DECIDEPMSG

function DECIDEPMSG
  $r^*_p \leftarrow \max_{B \in S} r_p(B)$
  $\overline{B} \leftarrow \arg\min_{B_j \in S, r_p(B_j) = r^*_p} L^r(j)$[8]
  Denote by $PR(\overline{B}, h, c)$ the proposal message about $\overline{B}$
  if $h = h_{root} + 1$ **then**
    if $B_{tc} = \varepsilon$ **then**
      $s_i \leftarrow \langle B_{root}, h_{root}, \varepsilon, 0\rangle$
      return $P^r_i(\overline{B}, h)$
    else
      if $r_p(\overline{B}) > F$ **then**
        $s_i \leftarrow \langle B_{root}, h_{root}, \varepsilon, 0\rangle$
        return $P^r_i(\overline{B}, h)$
      else if $\exists B' \in S$ s.t. $B' = B_{tc}$ **and** $r_p(B') \geq F$[9] **then**
        $s_i \leftarrow \langle B_{root}, h_{root}, B', r_p(B')\rangle$
        return $P^r_i(B', h)$
  return null

others can verify it by checking whether $e(G, S_i)$ is equal to $e(V_i, H(M))$ with a given bilinear map e. To track which signatures we have received, we append an integer array n of size N to the signature, and by signing a message M, a player computes $S_i = H(M)^{x_i}$, and increments the i-th element of n_{S_i}. The combination is the signature for aggregation, and we denote this process by $sign_i(M) = (S_i, n_{S_i})$.

An important property of the aggregated signature is that we can put in new signatures in an arbitrary order, avoiding the risk of adaptive chosen-player attack that Byzcoin[32] faces. Aggregating signatures is simply multiplying the BLS signature and adding up the array n. Thus, the aggregated signature (aka *multi-signature*) is $S = H(M)^{\sum_i x_i \cdot n_S[i]}$. We denote the process by $aggregate(S_1, S_2, ...) = (S, n_S)$. Let (S_1, n_{S_1}) and (S_2, n_{S_2}) be two multi-signatures, we can combine them by computing $aggregate(S_1, S_2) = (S_1 * S_2, n_{S_1} + n_{S_2})$. The array n tracks who have signed the message. Everyone can verify the multi-signature by checking whether $e(G, S) = e(\prod_i V_i^{n_S[i]}, H(M))$.

[40, 6] points out that aggregating signatures of the same message can be vulnerable to chosen-public-key attack. This attack can be avoided if the participates can prove they have the private key to their announced public key, either forced by a trusted third party or by a zero-knowledge-proof bootstrap process proposed by [40]. We choose this method because it's acceptable with the help of PKI.

Another method proposed in [6] computes $H(M + V_I)$ instead of $H(M)$ so each player signs different messages. Since everyone knows each others' public keys, the result is still verifiable without increasing the data size. This method does not involved a trusted third party or online bootstrap process, but it forces the algorithm to compute the bilinear map N times, instead of one time when the messages are the same. It can cause the verification time 100 times slower, and thus can only be adopted when the number of players in a system is small (less than 200).

The signature aggregation process significantly reduces memory utilization. Although the multi-signature still has size $O(N)$ asymptotically, a 4-byte integer is enough for each element in n_S in most cases. With 1,000 players using a 2048-bit signature, naively it takes 256 KB to store these signatures, but with aggregation, it requires only 4256 bytes, or 1/60 of the original size. The

optimization is more efficient as the system scales larger. For the case when the number of signers is small, the array is sparse and thus easily compressible.

5.7 Player Recovery from Temporary Failures

In normal cases, blocks and signatures are broadcast to all players. If a player misses a block in round r due to temporary failures, it can catch up in subsequent rounds using the following (offline) recovery procedures:

If player p_i receives a valid signature of enough signers in round r but fails to receive the block itself, p_i will check the signature and try to contact someone who has signed the block to retrieve it.

If player p_i recovers from an extended crash period and/or data loss, it should try to retrieve all lost blocks and proofs. She can only continue participating in the protocol after she recovers the entire history. As the committed blocks are offline-verifiable, blocks with valid signatures from any player are sufficient for recovery.

5.8 Security Analysis

We can prove that Gosig provides safety (as defined in Section 3) in fully asynchronous networks. Adding *partial synchrony* assumption, it also achieves liveness. We only list some key lemmas here and leave the complete proofs in Appendix A and B.

Lemma 1. *If an honest player p_i commits a block B at height h in round r, no player will ever TC any other block B' at any height $h' \leq h$ in any later rounds.*

Proof sketch. At least $f+1$ honest players will not P any blocks whose proposal rounds are no larger than r (line 14 to 20 in Alg. 2). Therefore, at least $f+1$ honest players will not P any other block at height $h' \leq h$ after round r, proving Lemma 1. And Lemma 1 leads to safety, because no block can be committed without honest players signing TC messages.

The following two lemmas prove the liveness under the partial synchrony assumption.

Lemma 2. *If in round r, for any honest player p_i we have $s_i = \langle B_{root}, h_{root}, \varepsilon, 0\rangle$, then there exists a round $r' > r$ and an honest player p_j such that p_j Ps some block at height $h = h_{root}+1$ in round r'.*

Lemma 3. *If in round r, there exists some honest player p_i with state $s_i = \langle B_{root}, h_{root}, B_{tc}, F\rangle$ $(B_{tc} \neq \varepsilon)$, then there exists a round $r' > r$ and an honest player p_j such that p_j commits a block at height $h = h_{root}+1$ in round r'.*

Attacks beyond the protocol layer. In addition to the adaptive chosen-player attacks and other attacks causing communication problems, an adversary can design attacks on the system implementations, including: 1) *computation resource saturation* attack, where the adversary may disseminate a large number of invalid messages to the honest players, consuming their CPU cycles for useless signature verification, and 2) *signature counter overflow* attack, where the adversary may craft valid multi-signatures where some counters are close to the maximum integer, and thus careless signature aggregating of honest players may cause an integer overflow resulting in incorrect signatures.

Note that both attacks can only cause liveness problems, rather than correctness problem. We will describe our countermeasures to both attacks in Section 6.

6 Implementation-level Optimization

As we mentioned, Gosig combines protocol level design and implementation level optimizations to achieve high performance. In this section, we introduce the important optimizations.

Asynchronous transaction dissemination and hash-only blocks. In our protocol, messages only contain *hashes* of the transactions to reduce message size. Raw transactions are gossiped among all players asynchronously, independent of protocol stages. In the case that a player does not have the raw transaction data when she receives the blocks, she retrieves the transactions from others before she can process the block. In practice, the gossip protocol often does a good job replicating the transactions, and thus this retrieval finishes fast.

Continuous gossiping in Stage II voting. As we need all honest players to receive proposed blocks and others' votes to achieve liveness, we do not limit the fanout. Instead, each player continuously sends block or P/TC messages to random neighbors until the end of the stage. However, we do put on a limit of concurrent connections to avoid overloading any player, which we set to 5 by default in our implementation. Section 7.3 provides a detailed analysis of the fanout limit.

Blacklisting obvious problematic players. While there is no way to ensure message delivery, if a player detects an obvious communication problem (connection failure, timeout etc.) with a peer, she will "blacklist" the peer (i.e. stop sending to it) for a short time period T_o (typically half a round time). On subsequent failure with the same peer, she will additively increase T_o, until she receives a message from that peer, or successfully retries. This backoff mechanism effectively limits the wasted attempts to connect to failed nodes.

LIFO processing stack. In Stage II, each node can concurrently receive multiple messages with signatures for processing (verification + aggregation). Sometimes the

messages arrive faster than the server can process them. We put them in a last-in-first-out (LIFO) *stack*, instead of a queue. This is because it is likely that later arriving messages contain more signatures, or sometimes even a super-set of signatures in earlier messages.

Preventing signature overflow. In the aggregated signature described in Section 5.6, we have the array n with N B-bit integers (we have $B = 32$ as default). An adversary can craft a valid signature so that the element corresponding to her signature is $2^B - 1$. This attack prevents honest players who have this signature from further aggregating the signature array because otherwise, the element will overflow.

We prevent such attack by restricting the growth of the maximum element in n, max, based on the number of signers s. On receiving a new message, the player tries to aggregate it into her local signature array, and check if the result satisfies $max \leq s$ or $\log_2(max) < B * s/N$. If so, the player updates her local array; otherwise, she drops the incoming message.

7 Evaluation

We evaluate the performance of Gosig using both simulations and real testbed experiments.

7.1 Evaluation Setup

Gosig prototype implementation. We implement the Gosig prototype in Java. We use `pbc` [39] library (with `JPBC` [19] wrapper) for cryptographic computation and use `grpc-java` [2] for network communication. As for signature parameters, we choose the default a-type parameter provided by `JPBC` [18], and use it to generate 1024-bit BLS signature. The entire system contains about 5,000 lines of Java code excluding comments.

Testbed. We build a testbed with up to 140 `t2.medium` instances evenly distributed on Amazon EC2's all 14 regions on 5 continents. We experimentally measure the network condition between the instances. Within a region, we have less than 1ms latency and about 100 MBps bandwidth, and latencies across regions are hundreds of milliseconds with the bandwidth varying from 2 MBps to 30 MBps. We believe the testbed is a good emulation of a typical multi-datacenter WAN.

Each instance acts both as client and server in the system. As the client, it generates transactions with an exponentially distributed inter-arrival time. Each transaction is 250 bytes, a typical Bitcoin transaction size (and used in evaluations of [41] too). These transactions are submitted to the servers on the same instance.

Simulation for larger scales. Limited by the testbed scale, we depend on simulations to analyze larger scale behavior of our signature collection process. We set the network latency to an exponential distribution with a mean of 300 ms, a typical value for today's Internet [36], and set the bandwidth to 500 KBps, a generous estimation after subtracting the bandwidth consumed by constantly gossiped transactions. We set the packet loss rate to 1% across all links (higher than many Internet links). In the simulator, we do not actually verify signatures, but set the signature verification time to be consistent with the performance we get on AWS `t2.medium` instances.[10] All faulty players in testbed experiments and simulation simply fail by crashing.

Key Configuration parameters. There are several configuration parameters to tune. The most important ones include the round time T and maximum block size `max_block_size`. Both are correlated and affect system scalability and performance significantly. We discuss their impacts in Section 7.4.

We set $T = 30$ sec (25 seconds for Stage I and 5 seconds for Stage II), and `max_block_size` = 8MB. Recall that in Gosig we only propagate transaction hashes (and send actual transactions asynchronously). Given that each transaction hash is 256 bit, a block can contain at most 250K transactions. With an average transaction size of 250 bytes, the corresponding raw transaction data for a block is about 62.5MB.

7.2 Real Testbed Performance

Here we present the performance metrics on the EC2-based testbed. We test two configuration settings, one optimized for throughput (the default setting), and the other optimized for transaction commit latency.

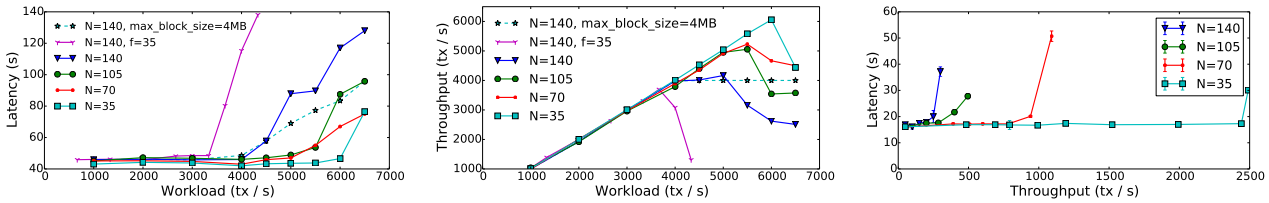
7.2.1 Throughput-optimized configuration

Using default parameter settings (Section 7.1), we run experiments on 35, 70, 105, 140 instances for 1200 seconds each using different workload from 1,000 tps to 7,000 tps. Figure 2(b) and 2(a) plot the throughput and average commit latency, respectively. We have the following observations:

1) Without overloading the system, the average commit latency is a little over 40 seconds. This is consistent with our theoretically expected latency of 1.5 rounds.

2) With 35 players, we can sustain a 6,000 tps workload. With 140 players, we can still support 4,000 tps. Comparing to the reported numbers in HoneyBadgerBFT [41] (using similar EC2 testbeds, but fewer geolocations), we double the throughput and reduce the latency by 80% with even more players and more regions.

[10] The verification time consists of an 11 ms constant overhead for computing bilinear map functions and another $0.11k$ ms for k signers. That means 12.1ms for 10 signers and 1111ms for 10000 signers.



(a) Latency with varying workload. (b) Throughput with varying workload (c) Latency vs. throughput

Figure 2: Performance under different configurations. (a) and (b) are throughput-optimized. (c) is latency-optimized.

3) When the system gets overloaded, the throughput actually *drops*. This is because, with a 30-second round time, there is not enough time to propagate all blocks to everyone, causing incomplete rounds and thus reducing the effective throughput (aka. goodput). To prevent such situation, we limit the `max_block_size` as an admission control mechanism, just like most blockchains do. In fact, the dashed line in Figure 2(b) shows that when limiting `max_block_size` to 4MB (i.e. 125K transaction per block, or 4167 tps), we can sustain the maximum throughput even on overloading. Of course, overloading still causes the latency to go up, but there is no difference from any queuing system.

4) Gosig tolerates failures quite well with small overhead. As Figure 2(a) and 2(b) shows, 35 faulty ones among 140 total nodes show little influence on the system's throughput or latency without overloading. The only impact of these failures is decreasing the maximal throughput by about 10%, from 4,000 tps to 3,600 tps.

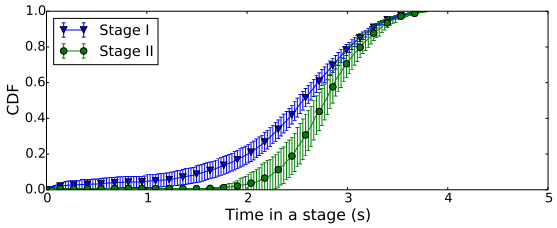


Figure 3: CDF for each player's stage completion times, using latency-optimized configuration. The error bars are 95% confidence intervals.

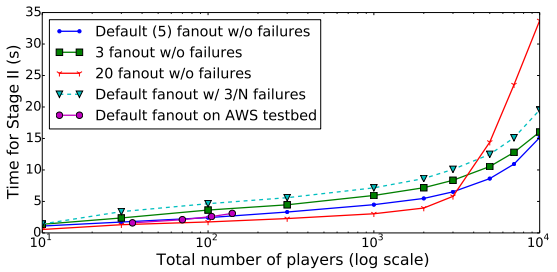


Figure 4: Simulation results for Stage II completion time.

7.2.2 Latency-optimized configuration

The default setup uses large block sizes and long round time (30 seconds) to improve overall throughput. For applications that are more latency-sensitive, we provide an alternative configuration. We reduce the round time T to 10 seconds with 5 seconds for each stage. We also disable block-existence probing (see Section 6) to further reduce latency. Then we repeat the same set of experiments, and Figure 2(c) shows the latency we can achieve under different workloads.

Like the previous case, we can get less-than-17-second latency and stable throughput until overloading. We can sustain over 200 tps with 140 nodes, 600+ tps with 70 nodes or 2,400+ tps with 35 nodes. However, the 10-second round offers very tight time budget for blocks to propagate. Larger blocks have little chance to complete propagation, causing the latency to go up quickly on overloading. Thus, a carefully controlled block size is even more essential.

In comparison, HoneyBadgerBFT cannot offer a low latency configuration in a relatively large group because of its $O(N^2)$-message-per-node complexity. Evaluation in [41] shows that the latency with only 64 players cannot even go below 200 seconds.

7.2.3 Latency breakdown

While the transaction commit latency is largely determined by the round time, we want to take a closer look at how fast a player can commit a block within a round. We plot the cumulative distribution (CDF) of the time taken for players to commit a block, using the same low-latency configuration with 140 nodes and 200 tps workload. Figure 3 shows the CDF of completion time for both stages on each node.

We can see that Stage II is only slightly slower than Stage I, especially at the slowest player (both at about 4 seconds). It is a little counter-intuitive as Stage II consists of 2 rounds of messaging (P and TC messages) vs. Stage I has only one. The reason is that to complete Stage I, every player has to receive the block from all leaders, in order to determine the least *leader score*. In comparison, Stage II only need votes from any $2f+1$ players.

7.3 Scalability

Limited by the resources on the testbed, we evaluate the scalability on systems larger than 140 nodes using simulation. In the simulation, we focus on the completion time of Stage II, because it is the core and most complicated part of Gosig, while the performance of Stage I is no different from other gossip-based systems.

Using the default settings, we show the time required to complete stage II with 10 to 10,000 players, i.e., all honest players receives a TC multi-signature signed by more than 2/3 players. Figure 4 shows the results using different combinations of failure modes and optimizations. The key observations are:

1) To calibrate the simulator, we also reproduce the testbed results (for up to 140 nodes) in Figure 4. We can see that it fits our simulation results quite well.

2) Gosig scales well. Even with 10,000 players, we can still finish Stage II within 15 seconds only. This is a direct benefit of using gossip-based application overlay network and asynchronous multi-signature, which fully exploits the redundant paths while keeping the bandwidth consumption small. The time grows faster when the number of players N is large, because the overhead for signature verification increases linearly with N. But this overhead only becomes significant after the total number goes beyond 3000, and can be reduces by stronger hardwares.

3) With 10000 players and 1/3 of them being faulty by crashing, Stage II completion time slowdowns from 14.97 seconds to 19.53 seconds. The robustness of the protocol comes from the gossip mechanism and the order-independent signature aggregation algorithm.

4) A small gossip fanout, i.e., the number of outbound connections, can fit most environments. A large fanout (like 20 in Figure 4) will saturate the network and cause higher latency due to queueing effects. Although a small fanout may not fully utilize the network when the system has fewer players, the cost is not significant since most time of a round is allocated to stage I.

7.4 Configuration Parameters

As we have seen in Section 7.2.1 and 7.2.2, `max_block_size` and round time T affect system performance significantly.

With N players, the `max_block_size` is proportional to three parameters [17, 47]: 1) $\frac{1}{\log N}$, 2) round time T, and 3) the network bandwidth. That means, in order to increase the number of nodes from 100 to 10,000, we need to either decrease the `max_block_size` by half or double the round time T given a fixed bandwidth.

In the remaining of the section, we experimentally evaluate their impacts using all 140 nodes in the testbed.

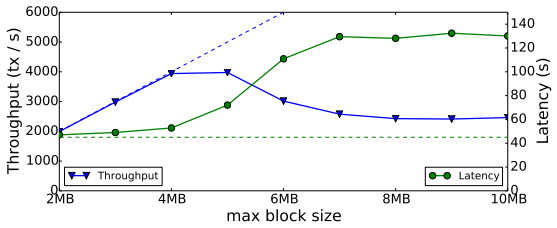


Figure 5: Influence of `max_block_size` on the performance.

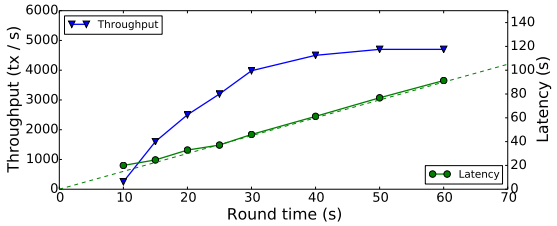


Figure 6: Influence of round time on the performance.

Max block size. As we have discussed, the parameter `max_block_size` serves as an admission control mechanism to avoid overloading the system.

Keeping $N = 140$ and $T = 30$s, we vary `max_block_size` from 2MB to 10MB, corresponding to 60K to 300K transactions per block, or 2K to 10K tps. In each round, we generate workload that equals the `max_block_size`. Figure 5 plots the results. The dashed line in Figure 5 shows the ideal case where the system has infinite capacity.

The actual throughput of the system is around 4,000 tps, as we presented in Section 7.2.1. We can see that for a `max_block_size` smaller than 4MB, the actual throughput increases with the `max_block_size` and roughly follows the ideal line. At around 4,000 tps, the system saturates. If the `max_block_size` is significantly larger than what the system can handle, the throughput decreases because some rounds will end before the players can fully propagate the blocks.

Round time T vs. throughput. Given a round time T, it is easy to calculate the expected transaction commit latency when the system is under a normal workload, which is $1.5T$. A latency significantly larger than this value indicates system overloading.

Here we experimentally find the maximum throughput we can obtain under different Ts, without overloading the system. Of course, because of the global clock synchronization error Δ and message propagation latency, we require T be at least 10 seconds.

Figure 6 plots both the throughput and latency under different T settings. We verify that we are able to keep the latency very close to the expected latency of $1.5T$. We observe that choosing a small T significantly reduces the maximum throughput. The throughput even drops

super-linearly as T gets smaller than 30 seconds. This is because when T and `max_block_size` are both small, the network setup overhead becomes non-negligible, further reducing the number of block transfers we can complete during the round. Fortunately, a T of 40 seconds already supports the max throughput in a 140-node system, still much faster than existing solutions.

8 Conclusion and Future Work

There are two types of approaches to build scalable blockchain systems: Some focus on theoretically provable protocols (e.g. HoneyBadgerBFT) even with high performance overhead, and others adopt best-effort hacks and hope it works most of the time (e.g. Bitcoin). We believe there should be a middle ground: we insist on a system with provable safety under a very strong adversary assumption, while adopting the best-effort approaches to increase the probability of staying alive. We use Gosig to demonstrate such a system. At the protocol level, using cryto-based secrete leader election and multi-round signature-based voting, we can guarantee safety, and by adding the partial synchrony assumption, we can also prove liveness. At the same time, we adopt implementation-level techniques such as gossiping, failure discovery and asynchronous signature verification to increase the probability of liveness. With evaluations on a real 140-server wide-area network, we show that Gosig both scales well and provides high performance.

As the next steps, we want to extend the protocol to allow players to join/exit. We also want to explore the approaches of integrating Gosig with other data-center-focused protocols to create a hybrid protocol, further improving its performance.

References

[1] Hyperledger consensus. https://github.com/diegomasini/hyperledger-fabric/blob/master/docs/FAQ/consensus_FAQ.md, 2016.

[2] grpc-java. https://github.com/grpc/grpc-java, 2017.

[3] AIYER, A. S., ALVISI, L., CLEMENT, A., DAHLIN, M., MARTIN, J.-P., AND PORTH, C. Bar fault tolerance for cooperative services. In *ACM SIGOPS operating systems review* (2005), vol. 39, ACM, pp. 45–58.

[4] BEN-OR, M., KELMER, B., AND RABIN, T. Asynchronous secure computations with optimal resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing* (1994), ACM, pp. 183–192.

[5] BIRMAN, K. P., HAYDEN, M., OZKASAP, O., XIAO, Z., BUDIU, M., AND MINSKY, Y. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS) 17*, 2 (1999), 41–88.

[6] BONEH, D., GENTRY, C., LYNN, B., AND SHACHAM, H. Aggregate and verifiably encrypted signatures from bilinear maps. In *International Conference on the Theory and Applications of Cryptographic Techniques* (2003), Springer, pp. 416–432.

[7] BONEH, D., LYNN, B., AND SHACHAM, H. Short signatures from the weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security* (2001), Springer, pp. 514–532.

[8] BONNEAU, J., MILLER, A., CLARK, J., NARAYANAN, A., KROLL, J. A., AND FELTEN, E. W. Research perspectives on bitcoin and second-generation cryptocurrencies. In *IEEE Symposium on Security and Privacy. IEEE* (2015).

[9] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 335–350.

[10] CACHIN, C. Architecture of the hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers* (2016).

[11] CACHIN, C., KURSAWE, K., PETZOLD, F., AND SHOUP, V. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference* (2001), Springer, pp. 524–541.

[12] CACHIN, C., AND VUKOLIĆ, M. Blockchains consensus protocols in the wild. *arXiv preprint arXiv:1707.01873* (2017).

[13] CASTRO, M., LISKOV, B., ET AL. Practical byzantine fault tolerance. In *OSDI* (1999), vol. 99, pp. 173–186.

[14] CLEMENT, A., KAPRITSOS, M., LEE, S., WANG, Y., ALVISI, L., DAHLIN, M., AND RICHE, T. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 277–290.

[15] CLEMENT, A., WONG, E. L., ALVISI, L., DAHLIN, M., AND MARCHETTI, M. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI* (2009), vol. 9, pp. 153–168.

[16] CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation 118*, 1 (1995), 158–179.

[17] CROMAN, K., DECKER, C., EYAL, I., GENCER, A. E., JUELS, A., KOSBA, A., MILLER, A., SAXENA, P., SHI, E., SIRER, E. G., ET AL. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security* (2016), Springer, pp. 106–125.

[18] DE CARO, A., AND IOVINO, V. Jpbc benchmark. http://gas.dia.unisa.it/projects/jpbc/benchmark.html, 2011.

[19] DE CARO, A., AND IOVINO, V. jpbc: Java pairing based cryptography. In *Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC 2011* (Kerkyra, Corfu, Greece, June 28 - July 1, 2011), IEEE, pp. 850–855.

[20] DECKER, C., SEIDEL, J., AND WATTENHOFER, R. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking* (2016), ACM, p. 13.

[21] DOUCEUR, J. R. The sybil attack. In *International Workshop on Peer-to-Peer Systems* (2002), Springer, pp. 251–260.

[22] EUGSTER, P. T., GUERRAOUI, R., HANDURUKANDE, S. B., KOUZNETSOV, P., AND KERMARREC, A.-M. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS) 21*, 4 (2003), 341–374.

[23] EYAL, I., GENCER, A. E., SIRER, E. G., AND VAN RENESSE, R. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), USENIX Association, pp. 45–59.

[24] GILAD, Y., HEMO, R., MICALI, S., VLACHOS, G., AND ZELDOVICH, N. Algorand: Scaling byzantine agreements for cryptocurrencies. SOSP 2017.

[25] GUO, Y., AND LIANG, C. Blockchain application and outlook in the banking industry. *Financial Innovation 2*, 1 (2016), 24.

[26] HIGGINS, S. Bitcoin mining pools targeted in wave of ddos attacks. https://www.coindesk.com/bitcoin-mining-pools-ddos-attacks/, 2015.

[27] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference* (2010), vol. 8, Boston, MA, USA, p. 9.

[28] KARP, R., SCHINDELHAUER, C., SHENKER, S., AND VOCKING, B. Randomized rumor spreading. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on* (2000), IEEE, pp. 565–574.

[29] KERMARREC, A.-M., MASSOULIÉ, L., AND GANESH, A. J. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed systems 14*, 3 (2003), 248–258.

[30] KIAYIAS, A., RUSSELL, A., DAVID, B., AND OLIYNYKOV, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference* (2017), Springer, pp. 357–388.

[31] KING, S., AND NADAL, S. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August 19* (2012).

[32] KOGIAS, E. K., JOVANOVIC, P., GAILLY, N., KHOFFI, I., GASSER, L., AND FORD, B. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)* (2016), USENIX Association, pp. 279–296.

[33] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 45–58.

[34] KURSAWE, K., AND SHOUP, V. Optimistic asynchronous atomic broadcast. In *International Colloquium on Automata, Languages, and Programming* (2005), Springer, pp. 204–215.

[35] KWON, J. Tendermint: Consensus without mining. *Draft v. 0.6, fall* (2014).

[36] LEDLIE, J., GARDNER, P., AND SELTZER, M. I. Network coordinates in the wild. In *NSDI* (2007), vol. 7, pp. 299–311.

[37] LI, H. C., CLEMENT, A., WONG, E. L., NAPPER, J., ROY, I., ALVISI, L., AND DAHLIN, M. Bar gossip. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 191–204.

[38] LIU, S., CACHIN, C., QUÉMA, V., AND VUKOLIĆ, M. Xft: Practical fault tolerance beyond crashes. *CoRR, abs/1502.05831* (2015).

[39] LYNN, B. On the implementation of pairing-based cryptography. *The Department of Computer Science and the Committee on Graduate Studies of Stanford University* (2007).

[40] MICALI, S., OHTA, K., AND REYZIN, L. Accountable-subgroup multisignatures. In *Proceedings of the 8th ACM conference on Computer and Communications Security* (2001), ACM, pp. 245–254.

[41] MILLER, A., XIA, Y., CROMAN, K., SHI, E., AND SONG, D. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 31–42.

[42] MILOSEVIC, Z., BIELY, M., AND SCHIPER, A. Bounded delay in byzantine-tolerant state machine replication. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on* (2013), IEEE, pp. 61–70.

[43] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system, 2008.

[44] PASS, R., AND SHI, E. Hybrid consensus: Efficient consensus in the permissionless model. In *LIPIcs-Leibniz International Proceedings in Informatics* (2017), vol. 91, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[45] SWANSON, T. Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems. *Report, available online, Apr* (2015).

[46] SYTA, E., TAMAS, I., VISHER, D., WOLINSKY, D. I., JOVANOVIC, P., GASSER, L., GAILLY, N., KHOFFI, I., AND FORD, B. Keeping authorities "honest or bust" with decentralized witness cosigning. In *Security and Privacy (SP), 2016 IEEE Symposium on* (2016), IEEE, pp. 526–545.

[47] TERRY, D. B., DEMERS, A. J., GREENE, D. H., HAUSER, C., IRISH, W., LARSON, J., SHENKER, S., STURGIS, H. E., AND SWINEHART, D. C. Epidemic algorithms for replicated database maintenance. *Acm Sigops Operating Systems Review 22*, 1 (1988), 8–32.

[48] TSCHORSCH, F., AND SCHEUERMANN, B. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Communications Surveys & Tutorials 18*, 3 (2016), 2084–2123.

[49] VASEK, M., THORNTON, M., AND MOORE, T. Empirical analysis of denial-of-service attacks in the bitcoin ecosystem. In *International Conference on Financial Cryptography and Data Security* (2014), Springer, pp. 57–71.

[50] VENKATARAMAN, V., YOSHIDA, K., AND FRANCIS, P. Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast. In *Network Protocols, 2006. ICNP'06. Proceedings of the 2006 14th IEEE International Conference on* (2006), IEEE, pp. 2–11.

[51] VERONESE, G. S., CORREIA, M., BESSANI, A. N., AND LUNG, L. C. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on* (2009), IEEE, pp. 135–144.

[52] VUKOLIĆ, M. The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. In *International Workshop on Open Problems in Network Security* (2015), Springer, pp. 112–125.

[53] WOOD, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper 151* (2014).

[54] ZHENG, Z., XIE, S., DAI, H.-N., AND WANG, H. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services* (2017).

A Safety

Lemma A.1. *If an honest player p_i commits a block B at height h in round r, no player will ever TC any other block B' at any height $h' \leq h$ in any later rounds.*

Proof. An honest player i commits a block B with a height of h means that at least 2f+1 players have *TC*ed B in round r, so at least f+1 honest players have *TC*ed B in round r. Those who commit the block B successfully will reject all messages for blocks of a height no higher than h, so we only consider the case where they fail to commit the block B.

There is no other block proposed before round r whose proposal round is larger than r. These f+1 honest players should have set their local $F = r$ according to line 16 in Algorithm 1, so they will not *P* any blocks proposed before round r.

Also, these f+1 honest player will not send *TC* message for any block of height less than h after round r, because they have committed

the same block of height $h-1$. Thus, no block of height less than h can be committed after round r, meaning no one is able to propose a new valid block with a height of $h' \leq h$ and with a proposal round larger than r after round r.

The only case left is that there can be more than one valid block proposed in round r, whose proposal rounds are the same with B's. In this case, by Algorithm 1, the $f+1$ honest players who have Ped B in round r will not P any other round-r block any more, so no other blocks proposed in round-r can be TCed, which requires P messages from at least $f+1$ honest players.

To sum up, at least f+1 honest players will not P any other block with a height of $h' \leq h$ after round r, and thus no one can TC any other block with a height of $h' \leq h$ after round r, which requires at least 2f+1 P messages. □

Lemma A.2. *If an honest player p_i commits a block B at height h in round r, no player will ever commit any other block B' at any height $h' \leq h$ in any later rounds.*

Proof. By Lemma A.1, we get that no one can TC any new block with a height of $h' \leq h$ after round r, so no honest player can commit any block with a height of $h' \leq h$ after round r, as he cannot collect enough TC messages. □

Theorem 1. *Gosig protocol achieves safety.*

Proof. Any committed block will be prepared by at least f+1 honest players. Since honest players will only P valid blocks, condition (1) of safety is true.

Condition (2) of safety can be directly proved by Lemma A.2. □

B Liveness

Since our potential leaders are elected secretly, the adversaries can not prevent honest players from becoming potential leaders of the least leader score, and can not prevent the block proposed by such an honest player from propagation since gossip peers are also secretly randomly chosen.

We base our proof on a partially synchronous network assumption. And when the network is synchronous, a transaction will fail to reach all honest players in a finite time with negligible probability. If there are always blocks packed by honest players committed, all transactions will be confirmed eventually.

Lemma B.1. *For any round r, there exists a round $r' > r$ and an honest player p_j such that p_j commits some block at height $h = h_{root} + 1$ in round r'.*

Proof. Without losing generality, we assume that at round r, the last committed B_{root} is the same for all honest players.

(Case 1) No player has TCed. Because the network will become synchronous infinitely, there exists a round where an honest player becomes the potential leader with the least leader score and the network is synchronous. In this round, the proposal round of the block proposed by this honest leader is equal to the local freshness F of all honest players, so it will be prepared and committed by all honest players.

(Case 2) Some player has TCed. Because at most one block can be TCed in a round, if two players TCed different blocks, these two players will have different local F. Thus, all players with the largest F will always have the same B_p, meaning if any of them becomes the leader in a synchronous round, this B_p will be prepared by all honest players and committed. Similar to Case 1, because the network will become synchronous infinitely, there exists a round where an honest player with the largest F becomes the potential leader with the least leader score and the network is synchronous, so its B_p can be proposed, prepared and committed.

To sum up, the lemma is proved. □

Theorem 2. *Gosig protocol achieves liveness.*

Proof. By Lemma B.1, the theorem is easily proved. □