Gosig: A Scalable and High-Performance Byzantine Consensus for Consortium Blockchains

Peilun Li
Tsinghua University
lpl15@mails.tsinghua.edu.cn

Guosai Wang
Tsinghua University
wgsjack199213@yeah.net

Xiaoqi Chen
Princeton University
xiaoqic@cs.princeton.edu

Fan Long
University of Toronto
fanl@cs.toronto.edu

Wei Xu
Tsinghua University
wei.xu.0@gmail.com

Abstract

Existing Byzantine fault tolerance (BFT) protocols face significant challenges in safety, scalability, throughput, and latency. We present a new BFT protocol, Gosig, for the consortium blockchains. Gosig guarantees safety even in asynchronous networks fully controlled by adversaries, by combining secret leader selection with multi-round voting. We co-design both the consensus protocol and the underlying gossip network to optimize performance. In particular, we adopt transmission pipelining to fully utilize the network bandwidth while use aggregated signature gossip to reduce the number of messages. These optimizations help Gosig to achieve unprecedented single-chain performance. On a public cloud testbed spanning multiple data centers consisting of 280 nodes across 14 cities on five continents, Gosig achieves over 15,000 transactions per second with 15.8-second confirmation time. When the system scales to 5,000 nodes, Gosig can still achieve 3,000 transactions per second with about 23.9-second confirmation time.

CCS Concepts

• **Security and privacy → Distributed systems security**.

Keywords

Byzantine fault tolerance, Blockchain, Scalability

ACM Reference Format:
Peilun Li, Guosai Wang, Xiaoqi Chen, Fan Long, and Wei Xu. 2020. Gosig: A Scalable and High-Performance Byzantine Consensus for Consortium Blockchains. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421272

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421272

1 Introduction

The rise of cryptocurrencies, such as Bitcoin [42], increases the awareness and adoption of the underlying *blockchain* technology. Blockchain offers a distributed *ledger* that serializes and records transactions. Blockchain provides attractive properties such as full decentralization, offline-verifiability, and most importantly, robust consensus at Internet scale. Thus, Blockchain has become popular beyond cryptocurrencies, expanding into different areas such as payment services, healthcare, and Internet-of-Things (IoT) [10, 51, 59].

While Nakamoto consensus is popular in applications of Bitcoin, it has severe performance limitations that hinder their further adoptions. For *consortium blockchains* deployed on traditional cloud services, where the identity of each node is known, it is more applicable to use Byzantine Fault Tolerant (BFT) protocols [13]. Recent work has shown that BFT algorithms can achieve roughly two orders of magnitude throughput improvement over Nakamoto consensus [23, 40]. However, traditional BFT protocols were initially designed for building replicated state machines on a small set of computers within a data center. Recent works [23–25, 33, 40, 57] try to adopt existing BFT protocols on blockchains by changing the network assumptions or reducing the total message sizes, but the following three significant challenges remain:

1. Targeted-single-point-failure is common in clouds with only a handful of data centers. It is easy for the adversaries to launch DDoS attacks targeting an honest node and partition it from others. Thus, any protocol relying on privileged publicly exposed nodes will lose liveness under so-called *adaptive attacks*.

2. The limited bandwidth and the long latency. The inter-data-center network has high latency and low bandwidth compared with the network in a data center, so most blockchain systems aggressively batch transactions into large blocks to amortize the consensus overhead (both bandwidth

and latency). While block broadcasting is often limited by network bandwidth, metadata dissemination is latency bound. Both are on the critical path, but the difference is often overlooked in existing BFT protocols. Some of them broadcast the block with a one-to-all pattern [13, 24, 36, 57], making the source a bottleneck. Some others broadcast the block with a simple gossip-like approach [25, 33], still leaving the broadcasting delay dominating the entire consensus process.

3. Limited by the slowest nodes. Both the communication and computation overheads on a single node limit the protocol scalability. And parties may use instances with different capacities/costs. If there is any node that needs to receive and process messages from all other nodes, it will become the bottleneck and slow down the entire process.

PBFT and its variants rely on a leader to order the requests. Since the cost to replace a leader with *ViewChange* is high and following leaders can be known and attacked in advance, they fail to solve challenges 1 and 3.

Most of the recent proposals, while solving challenge 3 relying on a small committee, fail to solve challenge 1. Algorand [23] avoids adaptive attacks with a combination of randomized committee election and a stateless BFT algorithm. However, it requires the network to stay synchronous periodically (a stronger assumption than PBFT) to recover from soft forks to guarantee safety.

Other attempts to solve challenge 3 is to use threshold-signature for message compression, like SBFT [24] or Hot-Stuff [57]. They reduce the total message size significantly. However, as the nodes need to wait until some collectors receive signature shares from all nodes, the performance is limited by their single-node capacity (challenge 3).

We present Gosig, a fast, scalable, and fully decentralized BFT system that solves the three challenges by integrating the protocol with the underlying gossip network.

Conceptually, Gosig operates in rounds, in each of which it appends one block (w.h.p.) onto the blockchain. Each round contains two stages, the block proposing stage and the signature collection stage. In the block proposing stage, Gosig first randomly selects several proposers. Then each proposer packs transactions into a new block and broadcasts the block to all other nodes. In the signature collection stage, each node chooses one block received in the first stage to vote, signs its decision, and keeps relaying the aggregated signatures signed by itself and received from the others. A node commits a block if it has collected enough signatures showing that no conflicting block can be committed.

Underlying the consensus protocol, Gosig uses a gossip network to propagate all messages. The consensus protocol ensures that messages in the same stage can be effectively aggregated during gossiping.

In addition to the common techniques adopted, Gosig adopts the following key: electing a random proposer for every new block, pipelining all possible processes, and aggregating all signatures on the fly using *gossip*-based broadcast.

Secure and cheap proposer replacement. To solve challenge 1, we change the proposers for every block. As a deterministically chosen proposer is vulnerable to adaptive attacks, we choose to implement the proposer election with a verifiable random function (VRF) [39] to make the selection random and unpredictable.

Besides, we also need to eliminate the expensive *ViewChange* in standard PBFT. We exploit the fact that a proposer proposes at most one block during its elected round, and achieves zero-overhead proposer replacement by including a small proof in every proposal to prove the new proposer's validity. The protocol proceeds in synchronous rounds, and at most one block can be committed in a round.

Aggregated signature gossip. To address the single-node capacity challenge, Gosig uses a novel aggregated signature gossip protocol to optimize the signature collection. It combines multiple received signatures into one equivalent aggregated signature [8] and relays the aggregated signature. We extend the multi-signature data structure [8] to allow arbitrary signing orders so that a set of aggregated signatures can be aggregated again during gossiping.

The signature aggregation makes the signature message size up to two orders of magnitude smaller and significantly reduces the total data transfer during the signature collection stage. More importantly, the reduction is achieved evenly among all participates, preventing any single-node capacity from becoming the bottleneck.

Transmission pipelining at all levels. We realize that after applying aggregated signature gossip, the vote exchanging is latency-bound, so a significant portion of the network bandwidth is under-utilized when everyone is waiting for enough votes to progress. Time is also wasted when everyone is waiting to hear a block proposal at the beginning of a round. Gosig pipelines at both the gossip layer and the BFT voting layer to maximize network utilization. We carefully design them to not affect protocol correctness.

We evaluate Gosig on an Amazon EC2-based 280-instance testbed that spans 14 cities on five continents. We can achieve a throughput of 15,000 250-byte transactions per second (tps) with an average transaction confirmation latency of 15.8 seconds, which is 8× the throughput and 1/10 of the latency comparing to HoneyBadgerBFT [40], a state-of-the-art protocol that offers the same level of safety guarantee. Also, using 1,000 AWS EC2 VMs to emulate 5,000 full nodes on a typical WAN bandwidth and latency, we show that Gosig can confirm 3,000 250-byte transactions per second with 23.9-second latency, achieving 3.6× throughput and 70% latency reduction compared to Algorand. And Gosig can also guarantee safety even under fully asynchronous network

(like traditional BFT). Our results demonstrate that the co-design of the BFT consensus algorithm and the Peer-To-Peer (P2P) network transmission layer can significantly boost the performance of large-scale BFT-based blockchain systems.

This paper makes the following contributions:

- **Gosig:** We present the design and implementation of Gosig, a BFT-based blockchain protocol. We show that by jointly optimizing the consensus and gossip protocols, we can achieve all the followings: 1) tolerance of *adaptive attacks* on the Internet, 2) full utilization of network resources, 3) scalability of existing blockchains (i.e., thousands of full nodes).
- **Aggregated signature gossip:** We present aggregated signature gossip, a novel technique that reduces the signature exchange traffic to 1/100 and significantly alleviates the message explosion problem when Gosig runs on thousands of full nodes.
- **Transmission pipelining:** We present transmission pipelining, a novel technique that jointly considers the communication patterns of BFT protocol and the gossip network, and achieves 7.5x higher throughput

2 Related Work

Bitcoin and its variants. Standard permissionless blockchains like Bitcoin [42], Ethereum [56] use proof of work (PoW) or proof of stake (PoS) to agree on a consistent transaction history to stop double-spending attacks. Combining with incentive mechanisms, they can prevent Sybil attacks and 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 [11, 48, 54] and use a BFT algorithm to achieve consensus on transaction history. Our focus is on the performance and safety of building a BFT-based blockchain 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 [14, 31, 36, 47]; 2) Using leader-less atomic broadcast: HoneyBadgerBFT [40] and [16, 32]. To prevent malicious leaders from affecting the system, Aardvark [15] use performance metrics to trigger view changes and Spinning [53], Tendermint [33] or others [5, 41] rotate leader roles in a round robin manner. However, these methods are vulnerable to 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 adopts a random proposer selection algorithm to keep safety and liveness under adaptive attacks.

SBFT [24] and Hot-Stuff [57] accelerate signature broadcasting with threshold signature, but the collectors or proposers still need to receive full-size signatures from *all* nodes before sending out the compressed signature, leaving themselves being the bottleneck. Hot-Stuff also avoids view-change by adding an extra signature exchanging phase, while Gosig chooses better normal-case performance over 'responsiveness' [57] in the worst case. ByzCoin [29] adopts CoSi [49] to reduce the pressure on the leader, but the tree structure and two-phase signing makes it vulnerable to adaptive attacks. However, in Gosig, by aggregating signatures during gossip, the signature size received by *every* node is reduced.

In order to scale the system, many systems adopt the "hybrid consensus" design [28, 29, 43, 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 their liveness or "optimal path". Algorand [23] leverages secret leader election and quorum member replacement methods to keep liveness but sacrifices safety by allowing temporary forks under asynchronous network. DFINITY [25] adopts *verifiable random function (VRF)* like Algorand, but only ensures safety under fully synchronous network. Stellar [37] allows participates to specify their trusted institutions individually and thus has a different trust model. In contrast, Gosig lets every node participate in the consensus and therefore ensures provable safety under asynchronous network.

OmniLedger [30] and RapidChain [58] choose a small committee for each shard to improve scalability, which can only be changed slowly (days) due to the reconfiguration overhead, and thus are still vulnerable to adaptive attacks.

Overlay network and gossip. Most consensus protocols use broadcast as a communication primitive. To improve the reliability on the Internet, people often use application-layer overlay networks. We adopt techniques like gossip from reliable multicast [7], probabilistic broadcast [22, 27] and other peer-to-peer (P2P) networks [26, 52]. Existing P2P networks may tolerate some Byzantine failures, but do not guarantee convergence [35]. By combining network optimizations like gossip with a robust protocol design, we can improve both system resilience and performance.

3 Overview

Gosig maintains a blockchain. Each node relays transactions to others and packs them into *blocks* in a specific order later. All committed blocks are serialized as a *blockchain*, which is replicated to all nodes. On the blockchain, one block *extends* another by including a hash of the previous block. A

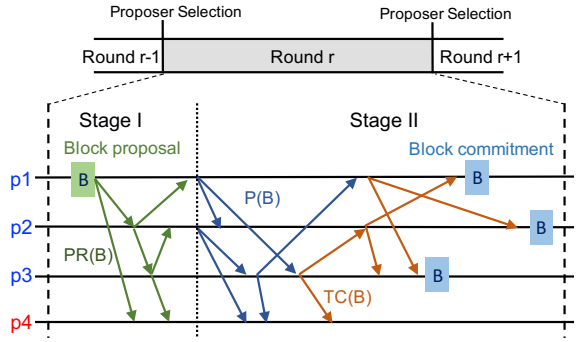


Figure 1: Overview of a Gosig round without pipelining (*happy path* only).

transaction is *confirmed* if and only if a block containing the transaction is *committed*. Gosig, as a consensus protocol, ensures that all blockchain replicas are the same.

3.1 System Model and Assumptions

Gosig has a standard system model and security assumptions commonly used in BFT protocols. These assumptions include: 1) Out of the $N = 3f + 1$ nodes in the system, at most f nodes have Byzantine failure, and the remaining $2f + 1$ work correctly; 2) nodes can authenticate the identities of other nodes through a trusted Public Key Infrastructure (PKI).

Note that Gosig has the following assumptions and guarantees: 1) we allow f faulty nodes to be corrupted by adversaries adaptively; 2) we guarantee safety in an asynchronous network and guarantee liveness in a *partially synchronous* [21] network, where all messages can be delivered within a known bound Δ after an unknown *Global Stabilization Time (GST)*; and 3) The liveness property also requires a loosely synchronous clock, which can be built with the partial synchrony assumption in theory [46] and is also provided by Network Time Protocol (NTP) in practice. Even if the clocks are arbitrarily deviated, the safety is not affected. Gosig also provides the *validity* property that all committed transactions follow some application-level predefined validation rules [33].

3.2 Gosig Protocol Overview

We provide an intuitive overview of Gosig in this section. We divide the execution of Gosig into *rounds* with a fixed time length. Each round consists of a proposer selection step (no communications) and two subsequent *stages* with a fixed length. Thus, all nodes know the current round number and stage by referring to the local clock.

Figure 1 provides an overview of a typical round. At the start of each round, some nodes secretly realize they are *potential proposers* of this round with a cryptographic sortition algorithm (Section 4.2.1). Thus, the adversary cannot target the proposers better than randomly guessing.

At Stage I, each selected potential proposer packs non-conflicting uncommitted transactions into a block proposal, disseminates it with gossip, and then acts just like a normal node afterward. The goal of Stage II is to reach an agreement on some block proposal of this round by vote exchange. A node "votes" for a block by adding its digital signature of the block to a *prepare* message ("P message") and gossiping it. An honest node only votes for a single proposal per round. Upon receiving at least $2f + 1$ signatures from P messages for a block, the node tentatively commits on this block by storing the messages and the round in its local state, and starts sending tentatively-commit messages ("TC message") for it. The node finally commits the block to its local blockchain replica once it receives $2f + 1$ TC messages. If it fails to finally commit the block in the same round, the local state will be used to decide whether to accept new proposals or decide which block to propose in the further.

We then optimize the protocol at every stage by applying the pipelining and aggregated signature propagation. See Sections 5.1 and 5.2 for the details.

4 Gosig Protocol Design

In this section, we first describe the sequential Gosig protocol (we call it the "vanilla protocol"). Then, in Sections 5.1 and 5.2, we discuss important optimizations, transmission pipelining and aggregated signature gossip, which enable Gosig to scale to thousands of nodes.

4.1 Message and State Definition

Section 3.2 briefly introduces the protocol, and we formally describe the notations in Table 1. We use $a.b$ to denote the field b of a structure a. For example, if c is a P certificate of the form $(H(B), h)$, $c.B$ denotes the block that c votes for.

Besides the blockchain replica with the proof for block commitment, each node also maintains a local state and decides its next actions based on the state and external events (receiving a certain message or selected as a proposer). The local state is also defined in Table 1. Note that $c_{tc}.r$ is always equal to r_{tc} if some block is TCed.

When Gosig starts running, the local state of a node is initialized as $s_i = \langle \epsilon, 0, \epsilon \rangle$ where ϵ stands for a null value. When a block is finally committed, this block will be appended to the blockchain and the state will be reset.

4.2 Stage I: Block Proposal

4.2.1 Proposer Selection We use *verifiable random function (VRF)* [39] to select a set of *potential proposers*. The idea is from Algorand [23]. We use a simplified version because we assume every node has the same weight, and the nodes cannot change their private keys after the system launches. The proposer selection protocol is described below.

Table 1: Notations used in the protocol description.

node p_i	the i-th ($i \in \{1, ..., N\}$)) node among all N nodes.
PR message	a `proposal` message of the form $(H(B), h, c, pp)$, where $H(B)$ is the hash of the proposed block B, h is the height of B, c is the *proposal certificate*, and pp is the *proposer proof*.
P/TC message	a `prepare`/`tentative-commit` message of the form $(H(B), h)$, where $H(B)$ is the hash of the voted block B, h is the height of B.
$PR_i^r(B)$	a PR message proposing a block B, signed by p_i in round r.
$P_X^r(B)/TC_X^r(B)$	an aggregated P/TC message voting for a block B, signed by a set of nodes X in round r.
p_i P/TCs B	p_i broadcasts a P/TC message about a block B.
P/TC certificate	an aggregated P/TC message signed by at least $2f + 1$ signers.
proposal certificate	a P or TC certificate c. Depending on the certificate type, the proposal certificate round $c.r$ is defined as the round in the P certificate, or the round in the TC certificate plus one.
node local state $s_i = \langle B_{tc}, r_{tc}, c_{tc} \rangle$	a 3-tuple where B_{tc} is the block itself, r_{tc} is the round that B_{tc} is TCed, and c_{tc} is the P-certificate used when B_{tc} is TCed in round r_{tc}. We use ϵ to represent a null value.

We define a publicly known random seed Q^h in Gosig 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[1], H is a secure hash function, l^h is defined as the proposer who has proposed the block B (the signer of the proposal certificate of B), and $SIG_i(M)$ is the Boneh–Lynn–Shacham (BLS) signature [9] of a message M signed with p_i's private key.

Based on Q^h, we define a node p_i's *proposer score* $L^r(i)$ at round r as $L^r(i) = H(SIG_i(r, Q^h))$, where h is the blockchain length at round r. If a node's *proposal score* is less than a *proposer threshold*, it becomes a *potential proposer*. At the beginning of a round r, each node p_i computes its $L^r(i)$, and knows if it is a *potential proposer* of the round. A potential proposer can prove to other nodes about its proposer status with the value $SIG_i(r, Q^h)$, called the *proposer proof*. The process requires no communication among nodes.

The cryptographic sortition algorithm has two properties: 1) the signature SIG_i uses p_i's private key, and thus cannot be forged; 2) If the hash function $H(\cdot)$ is perfectly random, the potential proposer 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 proposer.

The value of *proposer threshold* will only affect the performance but not the security of our protocol. Since the 'losing' blocks will be dropped fast as described in Section 5.1, the performance is not sensitive to the threshold as long as at least one proposer exists in most rounds. We set the threshold to $7/N$ where N is the total number of nodes, which is sufficiently large to reduce the probability of no-proposer rounds to less than 0.1%.

4.2.2 Block Proposals Each potential proposer p_i decides which block to propose and then generates a proposal message. If p_i has TCed a block B_{tc}, B_{tc} will be proposed. Otherwise, p_i generates a new block following the blockchain.

[1] Q^0 is a random number shared among all nodes.

To make a proposal valid, a potential proposer provides a *proposal certificate* c for the proposed block B. If B is chosen as the tentatively-committed block B_{tc}, the proposal certificate will be a P-certificate. Otherwise, if B is newly constructed, the proposal certificate will be a TC-certificate that commits the last block in the blockchain. This proposal certificate round $c.r$, as defined in Table 1 will be used to persuade other nodes who have TCed an earlier block to reset their state and prepare B. The certificate is signed by at least $2f + 1$ nodes so can not be forged.

Finally, the potential proposer p_i assembles a PR message as defined in Table 1. Then p_i signs the message with its private key. Everyone can easily verify the validity of the PR message by checking the included block, signatures and certificates. When a node receives a proposal, it will check if the height matched its local committed blockchain. If some committed blocks are missing in a node, the node will only vote until the missing blocks are recovered asynchronously as described in Section 5.3.

During Stage I, PR message are broadcast to all nodes by gossiping. At the end of Stage I (after a fixed time interval T_1), most nodes should have seen all block proposals in the round, assuming everything goes well. Nevertheless, Stage II is able to handle all complicated situations.

4.3 Stage II: Signature Collection

The objective of Stage II is to disseminate signed messages of nodes' votes for the block proposals. Same as Stage I, we use gossip to propagate all messages.

Stage II protocol. Algorithm 1 outlines the expected behavior of an honest node 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 its current local state and the incoming messages.

Lines 1 to 6 describe the initialization procedure, in which p_i checks all block proposals it receives in Stage I and decide

Algorithm 1 Stage II workflow for each node p_i.

State Variables:
– s_i: p_i's local state, i.e., $\langle B_{tc}, r_{tc}, c_{tc}\rangle$
– r: the current round number
– S: the set of valid proposals received in Stage I in round r

1: $phase \leftarrow$ Init
2: $\overline{B} \leftarrow$ DecideBlock ▷ See Algorithm 2
3: **if** $\overline{B} \neq null$ **then**
4: $SigP \leftarrow P_i^r(\overline{B})$
5: $SigTC \leftarrow null$
6: $phase \leftarrow$ Ped
7: **for** still in round r **do**
8: Keep gossiping the latest $SigP$
9: **if** $phase$ = TCed **or** Ced **then**
10: Keep gossiping the latest $SigTC$
11: **on** receiving a valid $P_X^r(B)$ message **do**
12: **if** $phase$ = Ped **and** $B = \overline{B}$ **then**
13: Update $SigP$ by aggregating $P_X^r(B)$
14: **if** $SigP$ is (2f+1)-signed **then**
15: Update $SigTC$ by aggregating $TC_i^r(B)$
16: $phase \leftarrow$ TCed ▷ Start gossiping $SigTC$
17: $s_i \leftarrow \langle B, r, SigP\rangle$
18: **on** receiving a valid $TC_X^r(B)$ message **do**
19: **if** $phase \neq$ Init **and** $B = \overline{B}$ **then**
20: Update $SigTC$ by aggregating $TC_X^r(B)$
21: **if** $phase \neq$ Ced **and** $SigTC$ is (2f+1)-signed **then**
22: $s_i \leftarrow \langle \epsilon, 0, \epsilon\rangle$
23: $phase \leftarrow$ Ced
24: Commit $\overline{B}$ with certificate $SigTC$

Algorithm 2 Decide which block to prepare in Stage II.

1: **function** DecideBlock
2: **if** $S = \phi$ **then** ▷ Received no valid proposals
3: **return** null
4: $\overline{z} \leftarrow \arg\min_{z \in S} L^r(j)$
5: **if** $B_{tc} = \epsilon$ **then**
6: $s_i \leftarrow \langle \epsilon, 0, \epsilon\rangle$
7: **return** $\overline{z}.B$
8: **else**
9: **if** $\overline{z}.c.r > r_{tc}$ **then**
10: $s_i \leftarrow \langle \epsilon, 0, \epsilon\rangle$
11: **return** $\overline{z}.B$
12: **else if** $\exists z \in S$ s.t. $z.B = B_{tc}$ **and** $z.c.r \geq r_{tc}$ **then**
13: $s_i \leftarrow \langle B_{tc}, z.c.r, z\rangle$
14: **return** B_{tc}
return null

which block to prepare (by calling the function *DecideBlock* in Algorithm 2). In general, p_i prefers a block proposal whose certificate round is larger, as it indicates a more recent block (lines 2.8 to 2.14). Finally, p_i chooses exactly one block $\overline{B}$ for height h and *P*s it (line 1.4).

After initialization, the state machine of p_i starts to handle incoming messages. Lines 11 to 24 in Algorithm 1 outlines handler routines for these two different message types. Node p_i only processes messages about the same block that it has *P*ed (line 1.12 and line 1.19), it can only *TC* the block $\overline{B}$ after it collects at least $2f + 1$ signatures from the *P* messages about $\overline{B}$, and it can only commit a block $\overline{B}$ after it collects at least $2f + 1$ signatures from the *TC* messages about $\overline{B}$ (line 1.14 and 1.21). These $2f + 1$ *TC* messages can be aggregated to a *commitment certificate* as an offline proof that $\overline{B}$ is committed.

Note that the local states will be reset either when blocks are committed or when all committed blocks are recovered. These rules ensure the safety and liveness of Gosig.

4.4 Security Analysis

We prove that Gosig provides safety in fully asynchronous networks. If we add a *partial synchrony* assumption (like PBFT [13]), it also achieves liveness. Due to limited space, we only list some key lemmas and proof sketches and leave the complete proof in a technical report [45].

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

Proof sketch. When p_i commits B, at least $f + 1$ honest nodes have *TC*ed B and set their $B_{tc} = B, r_{tc} = r$. Let this set of honest nodes be H_B. All nodes in H_B have committed some block at height $h - 1$ so they will not *P* any block at height $h' < h$ after round r. Thus, a block at height $h' < h$ cannot have $2f + 1$ *P*-messages to build a *P*-certificate and cannot be *TC*ed. Now we only need to prove the case $h' = h$.

Assume another block at height h is *TC*ed at round $r' > r$. It is *P*ed by $2f + 1$ nodes, so some node in H_B must have *P*ed it. According to Algorithm 2, a node with $B_{tc} \neq \epsilon$ will only change its B_{tc} state and *P* another block at height h if the block's proposal certificate c' has $c'.r > r$. Let us consider the first round that a proposal certificate c' with $c'.r > r$ is constructed for block B', $B' \neq B$. At round $c'.r$, all nodes in H_B have not changed their B_{tc} to another block at height h, they can only be in two cases: 1) have committed a block at height h, 2) still tentatively committed to block B with $B_{tc} = B$. In either case, they will not prepare block B' in round $c'.r$. Thus, without the votes from H_B, B' cannot get enough *P*-messages to construct a valid *P*-certificate in this round, and then cannot be *TC*ed or get a *TC*-certificate in this round. This contradicts the assumption that a proposal certificate for $B' \neq B$ is constructed in this round.

Now we have proved that no proposal certificate exists to persuade the nodes in H_B to prepare another block at height

h after round r. This means no other block at height h can have a P-certificate, and thus no node will ever TC another block at any height $h' \leq h$ after round r. □

Lemma 1 shows that after a block is committed, no other block at the same height can be committed. Thus, we achieve safety. The following lemma proves that liveness cannot be blocked forever if we add a partial synchrony assumption.

Lemma 2. *After Global Stabilization Time (GST), the probability to commit a block in a round is at least* $1/N$.

Proof sketch. After *GST*, if the proposer with the least proposal score is honest and its proposal certificate r_{tc} is no less than that of the other $2f$ honest nodes, its proposed block will be committed in this round. The probability that the honest node which enters a new round with the largest r_{tc} has the least proposal score is $1/N$. The selection is random and cannot be predicted by the adversaries, and the r_{tc} used as the proposal certificate round will not be affected by other nodes after entering a new round. Thus, the probability of committing a block in a round is at least $1/N$. □

Note that this lemma is for the worse case where only one node has the certificate with the largest round and only $2f + 1$ nodes are alive. After a successful round we should be able to make progress in every synchronous round later.

5 Key Performance Optimizations

5.1 Transmission Pipelining: Challenge 2 Solution

Gosig adopts transmission pipelining to optimize the communication patterns of the vanilla BFT protocol, and utilizes the network resources with higher efficiency. It allows independent block and message exchanges running asynchronously whenever possible. The goal is to minimize the waiting time of any node and utilize all idle bandwidths so that we can solve challenge 2.

Pipeline-friendly block structure. Gosig divides each block into the following three components: 1) the *header* including all metadata used in the protocol, such as the credentials and the proposal certificate, 2) the *body* containing an ordered list of transaction hashes, and 3) raw transactions in the block. The intuition behind this design is that the BFT protocol needs different kinds of information at different stages. Adopting this block structure enables Gosig to pipeline different stages of the protocol as much as possible and thus independently optimize the latency-sensitive metadata and bandwidth-bounded data transfers. It also enables Gosig to fully exploit independence among computation and data transfer by gossiping raw transactions and voting messages asynchronously.

Gosig further divides the body into several *segments*, allowing the nodes to start sending out data before receiving an entire block body, much like how BitTorrent transfers files. Different from BitTorrent, the proposer who generates the block signs each individual segment before gossiping them to different peers. Note that the proposer's signature at each segment is crucial because otherwise, the adversaries can generate lots of garbage segments to congest the entire network.

Pipeline between stages. The gossip network of Gosig first propagates the header of a block before propagating the body and its raw transactions. This enables the two stages, Stage I for block propagation and Stage II for signature propagation, to overlap with each other. Recall that the Stage I end time is actually a timeout when we can assume that all nodes have received the block with the least proposer score, so they can decide which block to P. In fact, a node only needs the block header to make the decision in Algorithm 2 (line 4). Thus, it is safe to propagate the headers first and move into Stage II as soon as all nodes have received the headers. As the header is typically small (about 41 KB for 10,000 participates, or 21 KB if we apply the compression discussed in Section 5.2), this optimization greatly shortens the Stage I time. Also, as nodes see the headers earlier, they can early decide the proposer with the least score without wasting bandwidths to propagate "losing" blocks from other proposers further.

Asynchronous raw transaction propagation. The gossip network of Gosig then propagates the body of a block with transaction hashes only. For the body, we adopt the idea of compact blocks from Bitcoin-core [1, 2]. We only include 6-byte short hashes instead of full 32-byte SHA256 hashes unless there is a collision. Therefore the body is also much smaller than all raw transactions combined.

Gosig gossips raw transactions asynchronously. When a node receives a transaction not previously seen, it passes the transaction to some random nodes (3 by default). If a node may see the hash-only block body before it receives the raw transactions, it retrieves missing transactions from random others before the node processes (i.e., tentatively commits) the block. To ask for these transactions, the node sends the 3-byte indices into the block instead of the 6-byte short hashes. As we can differentially encode [2] these indices, the average request size can be as small as 1 byte per missing transaction.

We still require nodes to receive the block body, gather all corresponding raw transactions, and validate them before they update their local states and send out the TC messages in Stage II (line 1.14 to 1.17 in Algorithm 1).

Pipeline between rounds. In the vanilla protocol, a round r starts only when Stage II of the previous round $r - 1$ ends, and potential proposers wait till the round starts to propose a block. We realize that as soon as a node commits a block in round $r - 1$ (i.e., see a message with enough signatures), the node can safely *start* round r by proposing a new block

for round r if it is a potential proposer. Nodes who are still in round $r-1$ will buffer these received blocks temporarily and process them either when they commit in round $r-1$ or when the starting time of round r comes.

Even if a node has entered round r before round $r-1$ ends, the node will keep doing its tasks for Stage II of round $r-1$, such as forwarding/signing P messages, until the end of the round. In this way, we can preserve the high signature propagation rate for round $r-1$.

Security analysis of pipelining. We now show that the pipelining design does not affect our consensus protocol. The *TC*-messages are only sent after validating the entire block, so all committed blocks are still valid.

A *P*ed block may become invalid for two reasons: 1) the block is reconstructed with the body but does not pass validity check, 2) the block is never fully reconstructed. In either case, the vanilla protocol will be able to commit a valid alternative block for the round, but the pipelined version commits none - as nodes early stopped propagating other candidates. This behavior is acceptable as the block can fail at the validation stage only if it is generated by a malicious proposer, who can stall the round anyways (by sending an empty block), i.e., we do not increase the power of an attacker.

Consider the following rounds. If a block is *P*ed but not *TC*ed, as Algorithm 2 indicates, the node state will only be affected by having B_{tc} reset. No information about this invalid block will persist. Thus, the next round will start as if a valid block is *P*ed but not *TC*ed. Now both safety and liveness of our protocol still hold, and pipelining does not give the adversaries any extra power.

5.2 Arbitrary-order Aggregated Signature Gossip: Challenge 3 Solution

Gosig uses aggregated signature gossip to optimize the signature transmission in Stage II of the vanilla BFT protocol, which reduces the data transfer for every node to avoid overloading any single node. Note that each node needs to collect votes or signatures from 2/3 of all other nodes. If naively implementing Stage II, the number of signature messages would grow super-linearly as the number of nodes. To address this scalability challenge, Gosig uses the techniques in [8, 9] to aggregate multiple received signatures into a compact *multi-signature* during gossiping. Specifically, Gosig collects signatures along with the gossip process by including *all* signatures a node has seen in the P message. In this way, on receiving a P message, a node can learn about multiple new signatures that it can *union* with those it has seen previously. This enables Gosig to forward one compact multi-signature instead of many individual signatures to save network resources. Also, accumulating signatures in messages instead of on nodes is more resilient to node failures. Since *every* node can send and receive less data, we can alleviate the single-node capacity problem expressed in challenge 3.

Now we describe how the signature aggregation works. The cryptographic signature of a node 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 node holding the private key x_i can sign a message M by computing $S_i = H(M)^{x_i}$, and 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 node computes $S_i = H(M)^{x_i}$, and increments the i-th element of n_{S_i}. The combination (S_i, n_{S_i}) is the signature for aggregation.

Aggregating signatures is simply multiplying the BLS signature and adding up the array n. Thus, the aggregated signature (aka *multi-signature*) is $S = H(M)^{\sum_i x_i \cdot n_S[i]}$. Let (S_1, n_{S_1}) and (S_2, n_{S_2}) be two multi-signatures, we can combine them by computing $(S_1 * S_2, n_{S_1} + n_{S_2})$. The array n tracks who have signed the message. Everyone can verify the multi-signature by checking whether $e(G, S) = e(\prod_i V_i^{n_S[i]}, H(M))$.

Note that the original signature aggregation algorithm [8] only allows collecting signatures in a particular order. We modify it by appending an integer array of size N to the signature to record how many times a node's signature is aggregated. This means we can aggregate multi-signatures in an arbitrary order [49], avoiding the risk of adaptive attack. [49] mentions the idea of appending arrays, but did not implement it, and thus vulnerable to adaptive attacks.

Although the multi-signature still has size $O(N)$ asymptotically, a 4-byte integer is more than enough for each element of the appended array. It only takes 40,256 bytes with 10,000 nodes using a 2048-bit signature, 0.64% of the size without aggregation. Also, as it is an integer array, Gosig compresses it effectively: 1) Gosig uses variable length integer encoding to reduce the size of many elements to 2 bytes; 2) when there are not many signers (most of the array elements are zeros), Gosig exploits the sparsity and encode it into a map; and 3) Gosig can further adapt integer array compression [34].

Continuous gossiping with LIFO processing stack. With the signature aggregation techniques, the gossip network now faces an interesting trade-off between forwarding a signature message immediately or waiting for more incoming signatures to exploit signature aggregation opportunities. In our implementation, each node continuously sends *P*/*TC* messages to random neighbors in Stage II under a limit of concurrent connections. The limit helps to avoid forwarding signatures too aggressively and losing the aggregation opportunities. The default limit is 5 and Section 6.4 provides a detailed evaluation of the setting.

If signatures arrive too fast to be processed in time, the node puts these messages into a *last-in-first-out (LIFO) stack*. Gosig chooses LIFO stack instead of a FIFO queue because it is likely that later arriving messages contain more signatures.

5.3 Handling Special Cases

Participants join / leave. We support any non-interactive proof authorizing a node to join or leave, be it a signature from a trusted authority or a multi-signature from a group. To join / leave the group, a node submits a special transaction containing the proof. Existing nodes will update their own participant list on committing the transaction.

Handling temporary failures. If a node recovers from a crash or data loss, it should retrieve lost blocks and proofs. It can only continue participating after it recovers the entire history. Since any voting of a block can only be processed after committing the previous block, all honest nodes will see the same membership when counting the votes.

If a node detects an obvious communication problem (connection failure, timeout, etc.) with a peer, it will "blacklist" the peer (i.e. stop sending to it) for a period T_o (typically a half of the round time). On subsequent failures with the same peer, it will additively increase T_o, until it receives a message from that peer, or successfully retries. This backoff mechanism effectively limits the wasted attempts to connect to failed nodes.

6 Evaluation

We evaluate the performance of Gosig using a combination of a real-world testbed, emulations, and simulations. We present both the overall performance and the effects of various parameters and optimizations. We also evaluate the effectiveness in solving the poor network condition and single-node capacity challenges using large-scale simulations.

6.1 Evaluation Setup

Gosig prototype implementation. We implement the Gosig prototype in Java. We use `pbc` [38] library for cryptographic computation (with `JPBC` [19] wrapper and with preprocessing enabled) and use `grpc-java` [3] 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 5,500 lines of Java code excluding comments.

We use three testbeds for different applications and scales, including a real 280-node inter-datacenter WAN, a 5K-node WAN emulator, and a 10K-node large-scale WAN simulator.

Workload. Each transaction submitted to Gosig is a simple key-value set operation. The transaction is padded to 250 bytes, a typical Bitcoin transaction size (and used in [40]). The transaction execution does not involve signature verification or disk IO operations, so our evaluation can focus on the performance of the consensus protocol. These transactions are submitted to the server running on the same EC2 instance. Note that the latency is measured end-to-end, between the moment a transaction is generated and the moment the block packing it is confirmed, while most other works [29, 30, 40] only take into account the confirmation latency of a block, not individual transactions. Theoretically, the expected *transaction commit latency* is about 1.5× of the *block confirmation latency*, because all transactions generated in a round are likely to be packed and committed in the next round.

Real 280-node inter-datacenter WAN testbed. We build a multi-region cloud testbed using 280 `t2.large` instances (2 cores, 8 GB memory) evenly distributed on Amazon EC2's 14 regions on 5 continents. We experimentally measure the network condition. Within one region, we observe <1ms latency and about 500 Mbps bandwidth, and latencies across regions are up to hundreds of milliseconds with the bandwidth varying from 15 Mbps to 250 Mbps. We believe this is a good representation of a multi-region cloud.

A 5K-node emulation. To evaluate the scalability of Gosig, we emulate a WAN with 5,000 nodes using 1,000 EC2 instances. The setup is similar to Algorand [23]. We run 5 nodes per EC2 `m4.2xlarge` instance. We limit the network bandwidth of each node to 20 Mbps. We insert artificial latency on the sender side to emulate the network delay, according to the measurements among 20 cities across the world [55].

To evaluate more nodes on each EC2 instance, we use sleep to replace the signature verification - the CPU heavy operation. We set the sleep time to the verification time we measured on `m4.2xlarge` instances[2].

A 10K-node simulation. Considering the cost, we use simulations to analyze Gosig on the public Internet. We focus on the core of the protocol - the signature collection process. We use the same network latency configuration and signature verification time as our emulation, set the network bandwidth limit to 2Mbps[3], and set the packet loss rate to 1% (higher than many Internet links).

6.2 Real 280-node Testbed Performance

We first present the performance comparison with Hyperledger Fabric [6], a well-known consortium blockchain system for production, on a small cluster. The comparison will show that with our setup, running a full-stack blockchain

[2]The verification time consists of an 11 ms constant overhead for computing bilinear map functions and another $0.008k$ ms for k signers with appended array elements less than 2^{16}.

[3]The bandwidth limit fits our results in Section 6.3 by subtracting the bandwidth for transaction and block propagation.

Table 2: Performance comparison of Gosig and Fabric

System	Latency (s)		Throughput (tps)	
	5 nodes	20 nodes	5 nodes	20 nodes
Fabric	4.44	4.84	198	187
Gosig	7.70	7.72	23500	21000

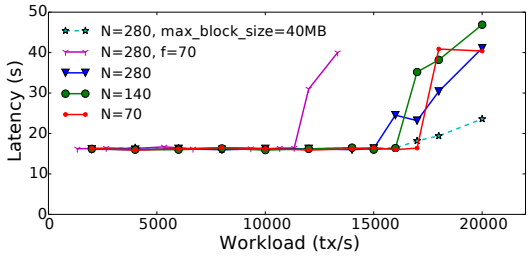
system cannot effectively evaluate the consensus protocol. We then present the overall performance of Gosig on the 280-node testbed and then provide detailed analysis of configuration parameters and the effects of the optimizations.

6.2.1 Comparison with Hyperledger Fabric We use up to 20 nodes of 5 regions from 5 different continents to run the experiment. On each node, we deploy both a peer and an orderer with the official docker image of version 2.2.0. The transactions submitted to Hyperledger Fabric are `EmptyContract` in Hyperledger Caliper [4], which is just a no-op. The transactions are evenly generated by all nodes, and they are submitted and endorsed by the peer deployed on the same node. The workload is similar to the one used in Gosig as they try to be as simple as possible. Fabric has not officially supported any Byzantine fault tolerant algorithms, so we choose Raft (`etcdraft`) as the consensus module. One round of Gosig is 5 seconds, and the batch timeout of Fabric is 1 second.

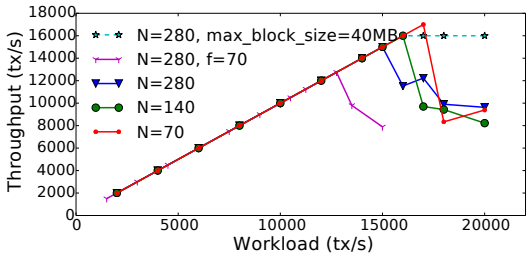
Table 2 shows the achieved throughput and latency. The throughput of Fabric is less than 1% of Gosig with our 2-core hosts because all CPU resources are occupied by the verification of the certificates and signatures of transactions. However, these transaction-related verifications are actually at the application level and are not a part of the consensus protocol that we want to evaluate. Besides, this overhead can be reduced by a more powerful machine because they can be processed in perfect parallelism, and this solution is independent of the consensus module. Thus, in all the following evaluations, we will only run experiments on Gosig with dummy transactions to show the performance of the consensus protocol.

6.2.2 Overall performance For the default experiments on the testbed, we set round time $T = 10$ sec (5 sec for each stage), and `max_block_size` = 60 MB. Note that we can support large block size because we use segments to pipeline block transfer and verification. With an average transaction size of 250 bytes, we can include up to 240,000 transactions per block. As we only include the block header in the proposals, the *PR* messages are much smaller than `max_block_size`. We leave the discussion of configuration choices to Sec 6.2.3.

We run experiments on 70, 140 and 280 instances (i.e., 5, 10, 20 instances per data center). We run each for 1,200 sec using different workload varying from 4,000 to 20,000 transactions per second (tps). Figure 2(b) and 2(a) plot the throughput and average commit latency, respectively. We have the following observations:



(a) Latency with varying workload.



(b) Throughput with varying workload

Figure 2: Performance under default configurations.

1) Without overloading the system, the average transaction commit latency is about 15.8 s, consistent with our theoretically expected latency of 1.5× round time ($T = 10$ s).

2) With 70 nodes, we can sustain a 17,000 tps. With 280 nodes, we can still support 15,000 tps, which clearly demonstrates the performance that Gosig can achieve. Comparing to the reported numbers in HoneyBadgerBFT [40] (using a similar EC2 testbed, but fewer geo-locations), we reach 8× the throughput and reduce the latency by 90% with more nodes and more regions.

3) When the system gets overloaded, the throughput actually *drops*. This is because the 10-second round time is not enough to propagate all blocks to everyone, causing incomplete rounds and thus reducing the effective throughput. To prevent such situation, we limit the `max_block_size` as an admission control mechanism. In fact, the dashed line in Figure 2(b) shows that when limiting `max_block_size` to 40MB (i.e., 160K transaction per block), we can sustain the maximum throughput even when overloaded. Of course, overloading still causes the latency to go up, the same for all queuing systems.

4) Gosig tolerates failures quite well with small overhead. As Figure 2(a) and 2(b) shows, 70 faulty nodes (not responding) among 280 total nodes show little influence on the throughput or latency without overloading. The only impact of these failures is decreasing the maximal throughput by about 20%, from 15,000 tps to 12,000 tps.

6.2.3 Configuration Parameter Choices Gosig offers several parameters to accommodate different system conditions. The

most important ones include the round time T, maximum block size `max_block_size`, and the number of segments per block. We experimentally evaluate their impacts on the same 280-node testbed.

Max block size. As we have discussed earlier, the parameter `max_block_size` serves as an admission control mechanism to avoid overloading the system. With N nodes, the `max_block_size` is proportional to three parameters [17, 50]: 1) $\frac{1}{\log N}$, 2) round time T, and 3) the network bandwidth. That means, 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.

Keeping the number of nodes $N = 280$ and round time $T = 10$s, we vary `max_block_size` from 20MB to 50MB, corresponding to 8K to 20K tps. In each round, we generate workload that saturates the `max_block_size`. Figure 3 plots the results. The dashed line in Figure 3 shows the ideal case where the system has infinite capacity.

The actual throughput is around 16,000 tps, as we presented earlier. We can see that for a `max_block_size` smaller than 40MB, the actual throughput of the system increases with the `max_block_size` and roughly follows the ideal line. At around 16,000 tps, the system saturates. If the `max_block_size` is significantly larger than the system's capacity, the throughput decreases because some rounds will end before the blocks can be fully propagated.

Round time T. The round time T directly impacts the commit latency. The expected transaction commit latency without overloading is $1.5T$ if we can commit all transactions generated in one round in the next. The latency significantly increases than this value, which indicates system overloading. We want to evaluate the influence of T settings. Note that `max_block_size`$/T$ (i.e., the data committed per round) is limited by the network bandwidth. Thus, for each T choice between 5 and 20 sec, we experimentally determine the corresponding optimal `max_block_size` that allows the maximum throughput. We equally split T between the two stages. We require T to be at least 3 sec to tolerate the clock synchronization error Δ and message propagation delay.

Figure 4 plots both the throughput and latency under different T settings. In the figure, 1) we verify that we are able to keep the latency close to the expected latency of $1.5T$; and 2) we observe that choosing a small T significantly reduces the maximum throughput. This is because when T is small, the corresponding `max_block_size` has to stay small, meaning we are propagating many small blocks in very short rounds. In this case, the network setup overhead becomes non-negligible, further reducing the bandwidth we can use to transfer useful data; 3) setting T to 10 sec already supports the max throughput (16,000 tps) in a 280-node system, much faster than all existing solutions to our knowledge.

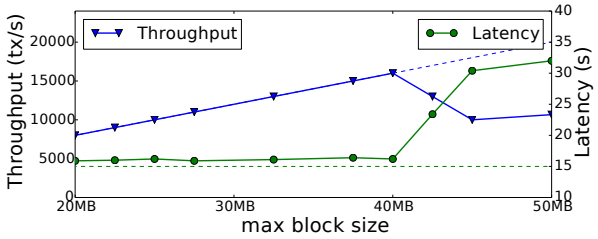


Figure 3: Performance effects of `max_block_size`

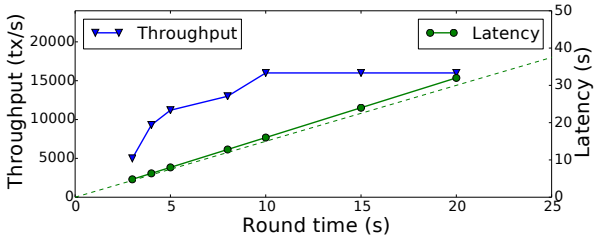


Figure 4: Performance effects of round time

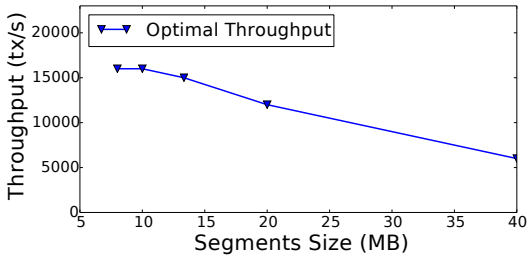


Figure 5: Performance effects of block segments

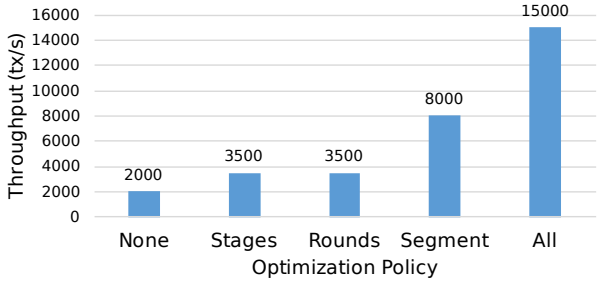


Figure 6: Throughput of different optimizations.

Number of segments per block. Figure 5 shows the throughput improvement by transferring blocks in segments. Using 13.3 MB segments in 40 MB block, we can achieve the throughput about 15,000 tps. This is because, with segments, the nodes who are close to the round proposer can start gossiping out the block body earlier, and thus accelerate block body propagation. Also, since nodes can retrieve missing transactions without receiving the entire block body, raw transactions also propagate faster. We also observe that we only need a small number of segments per block to saturate network bandwidth in our testbed.

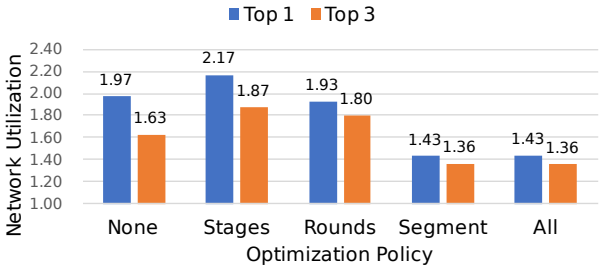


Figure 7: Uplink network utilization of the top hotspot nodes, normalized by dividing the mean.

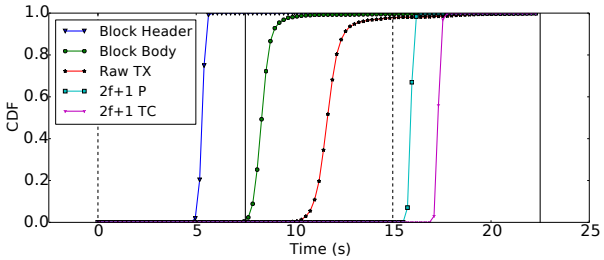


Figure 8: CDF for each stage time with 5k nodes.

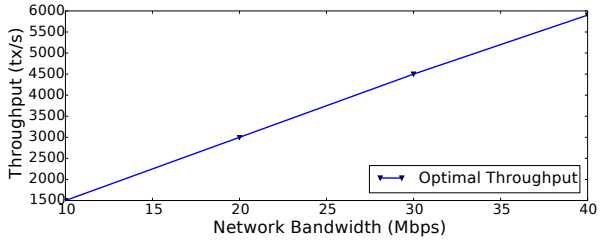


Figure 9: Maximum throughput supported by limited network bandwidth from 10Mbps to 40Mbps.

6.2.4 Benefits of Pipelining We evaluate how each optimization in Secion 5.1 improves performance. We keep the standard optimizations, like short hash, and measure the performance by selectively disabling other optimizations. For all experiments, we keep round time $T = 10s$ and experimentally determine the optimal `max_block_size`.

Figure 6 shows that the vanilla implementation with all pipelining disabled can only achieve 2,000 tps. Pipelining between round or stages can both increase the throughput by about 75% to 3,500, because it allows block propagation to overlap with signature propagation, increasing the network utilization of all nodes in Stage II. Using 5 segments per block, we improve the throughput from 2,000 tps to 8,000 tps. Adding all the improvements together, the throughput can reach 15,000 tps, or 7.5 × better than the vanilla protocol.

Load balancing across nodes. The performance improvements mainly come from reduced waiting time, which is the result of a much more balanced network utilization among the nodes. Figure 7 plots the relative network traffic at the top 1 and top 3 busiest nodes to the average traffic across all nodes. We can see that using segments is the most effective in reducing hot spots. This is because more nodes can start propagating before receiving the full block, reducing the waiting for the first block.

6.3 5K-node Emulation

Overall performance. We evaluate the scalability of Gosig with the 5K-node emulation. We set the round time to 15 sec (evenly split between two stages) and the `max_block_size` to 12 MB. We set the workload to 3000 tps, sufficient to provide the optimal throughput. Under this configuration, we can confirm 2700 MB data per hour, with only 23.9 sec transaction commit latency.

The emulation shows that Gosig achieves both higher throughput (about 3.6×) and shorter confirmation time (70% less) compared with the result presented by Algorand [23], which commits 750 MB data per hour with about 50 sec for each round (or 75 sec transaction commit latency if the workload is generated uniformly) [4]. Moreover, Gosig runs consensus among all 5,000 nodes, while Algorand only uses a committee of 2,000 nodes.

Each round is fully utilized. The good performance partially comes from the fact that transmission pipelining can overlap the waiting time of different messages and utilize the synchronous rounds in an asynchronous way. Figure 8 illustrates the timeline how a block propagates in a round r. It plots the percentage of nodes who have received the block header (from the winning block), the entire block body, all raw transactions in this block, and enough $(2f + 1)$ P/TC signatures, respectively. The solid vertical line separates different rounds, and the dashed vertical line separates stages within a round. We start plotting at the beginning of Stage II of round $r - 1$.

The potential proposers of round r start to propose blocks (and send the headers) when they commit a block in round $r-1$, about 2.5 sec before round r starts as we pipeline between rounds. Starting a round early provides us the flexibility to adjust the workload among different rounds, to better handle both workload variations and performance disturbances.

It takes only about 2 s for all nodes to receive the header of the proposed block, thanks to its small size. The round r officially starts at time 7.5 sec, but this start time is not important in normal cases, because all nodes have already committed round $r - 1$ and entered round r before this time. When the Stage II of round r begins (at 15s), a small fraction of nodes has not received all the transactions of the proposer block. They can send their P votes now since they have

[4] Algorand runs 50K users to get the metric. For 5K users, Algorand only shows the throughput of about 200 MB per hour with 18-second rounds.

received the header, but they have to receive these missing transactions in Stage II before sending out their TC votes. This pipelining between stages allows Stage II to start early and makes the stage splitting configuration less sensitive.

Network bandwidth is fully utilized. To show that Gosig can also utilize the bandwidth well, we change the network bandwidth limit from 10Mbps to 40Mbps, and Figure 9 shows the maximum throughput. We see that the throughput is almost proportional to the network bandwidth (1500 tps for 10Mbps and 5900 tps for 40Mbps). We analyze the log and notice the actual bandwidth utilization for raw transaction transmission (including both gossip and missing transaction retrieval) is 8.25 Mbps, 16.68 Mbps, 25.01 Mbps, 32.70 Mbps for 10, 20, 30, 40 Mbps bandwidth respectively. Over 80% bandwidth is devoted to 'useful' payload transmission, meaning our protocol introduces little overhead and utilizes the bandwidth well.

6.4 10K-node Simulation

Our 10K-node simulation focuses on the completion time of Stage II, the core and most complicated part of Gosig.

Under the same configurations as in the emulator, we show the time required to complete stage II with 100 to 10,000 nodes, i.e., all honest nodes receive a *TC* signed by more than 2/3 nodes. Figure 10 shows the results using different combinations of failure modes and optimizations. There are several observations:

1) (Simulator calibration) We first reproduce the 5K-node emulation result. The star in Figure 10 shows time take to complete Stage II in the 5K-node emulation. We find that the time matches the simulator very well: 3.55 sec in the emulation and 3.70 sec in the simulation. It shows that the simulator does reflect the actual performance.

2) Gosig scales well. Even with 10,000 nodes, Stage II only takes 6.68 sec. Multi-signature plays an important role with a large number of nodes. Without multi-signature[5], it takes 4× more time to finish Stage II.

3) With 10,000 nodes and 1/3 of them not responding, Stage II completion time slows down from 6.68 sec to 11.31 sec. The robustness comes from the gossip mechanism and the order-independent signature aggregation.

Aggregated signature gossip solves the single node capacity challenge. Figure 10 also provides a comparison between threshold signatures and aggregated signature gossip. Threshold signatures [24, 57] behaves well at the scale of hundreds. However, when the scale exceeds 1,000, the latency increases rapidly, and with 10k nodes, it gets over 20 sec. The performance drops because the message size processed by the collector grows linear with the group size, making this single node slow down the whole system. In

[5]We use 1024-bit RSA signature that offers the same security level.

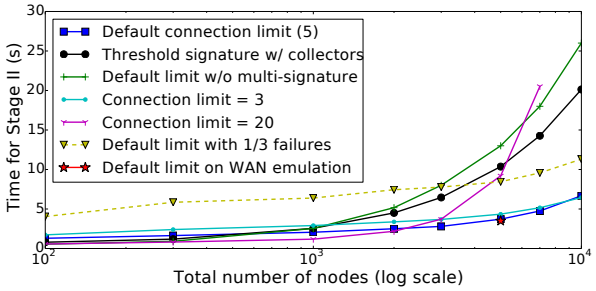


Figure 10: Completion time simulation for Stage II

contract, aggregated signature gossip compresses the data sent and received by every node and significantly boosts the performance of Stage II. When Gosig runs with 10k nodes, Stage II takes only 6.68 sec to complete.

Our results also show that a small gossip connection limit, i.e., the number of outbound connections, can fit most environments. A large limit (e.g., 20 in Figure 10) will saturate the network and miss signature aggregation opportunities. Although a small limit may not fully utilize the network with fewer nodes, the cost is not significant compared with time already spent on block propagation.

7 Conclusion and Future Work

While BFT protocols are a promising tool to build consortium blockchains, they face significant challenges when adopted into cloud services on the public Internet. We present a new BFT protocol, Gosig, for consortium blockchains. In Gosig, proposers for every new block are randomly and secretly elected to prevent adaptive attacks. We jointly optimize the BFT protocol layer and the gossip layer, pipelining all communications to maximize the network utilization wherever possible. Using aggregated signature gossip, we can pack the partial voting results into a short message, allowing every single node to transfer fewer data. These optimizations help Gosig to support a single consensus group with thousands or even more nodes, and achieve several times higher performance in a wide area network while maintaining the same level of safety as traditional BFT (e.g., tolerate asynchronous network and adaptive attacks).

As future work, we would like to design an incentive mechanism to encourage players to follow the protocol, e.g., does its best to forward messages. Also, Gosig is both a complete system and a building block for advanced blockchains. For example, we would like to provide a sharding/off-chain design using Gosig as consensus protocols for each shard.

Acknowledgments

This work is supported in part by the National Natural Science Foundation of China (NSFC) Grant 61532001 and the Zhongguancun Haihua Institute for Frontier Information Technology and Nanjing Turing AI Institute.

References

[1] 2016. Bitcoin Core. https://bitcoincore.org/.

[2] 2016. Bitcoin Core BIP 152. https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki.

[3] 2017. grpc-java. https://github.com/grpc/grpc-java.

[4] 2020. Hyperledger Caliper. https://github.com/hyperledger/caliper.

[5] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. 2005. BAR fault tolerance for cooperative services. In *ACM SIGOPS operating systems review*, Vol. 39. ACM, 45–58.

[6] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.

[7] Kenneth P Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. 1999. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)* 17, 2 (1999), 41–88.

[8] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. 2003. Aggregate and verifiably encrypted signatures from bilinear maps. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 416–432.

[9] Dan Boneh, Ben Lynn, and Hovav Shacham. 2001. Short signatures from the Weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 514–532.

[10] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. 2015. Research perspectives on bitcoin and second-generation cryptocurrencies. In *IEEE Symposium on Security and Privacy. IEEE*.

[11] Christian Cachin. 2016. Architecture of the Hyperledger blockchain fabric. In *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*.

[12] Christian Cachin and Marko Vukolić. 2017. Blockchains Consensus Protocols in the Wild. *arXiv preprint arXiv:1707.01873* (2017).

[13] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[14] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 277–290.

[15] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *NSDI*, Vol. 9. 153–168.

[16] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. 1995. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation* 118, 1 (1995), 158–179.

[17] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. 2016. On scaling decentralized blockchains. In *International Conference on Financial Cryptography and Data Security*. Springer, 106–125.

[18] Angelo De Caro and Vincenzo Iovino. 2011. JPBC Benchmark. http://gas.dia.unisa.it/projects/jpbc/benchmark.html.

[19] Angelo De Caro and Vincenzo Iovino. 2011. jPBC: Java pairing based cryptography. In *Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC 2011*. IEEE, Kerkyra, Corfu, Greece, June 28 - July 1, 850–855. http://gas.dia.unisa.it/projects/jpbc/

[20] Christian Decker, Jochen Seidel, and Roger Wattenhofer. 2016. Bitcoin meets strong consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*. ACM, 13.

[21] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[22] P Th Eugster, Rachid Guerraoui, Sidath B Handurukande, Petr Kouznetsov, and A-M Kermarrec. 2003. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems (TOCS)* 21, 4 (2003), 341–374.

[23] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine agreements for cryptocurrencies. SOSP 2017.

[24] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: a scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 568–580.

[25] Timo Hanke, Mahnush Movahedi, and Dominic Williams. 2018. DFINITY Technology Overview Series, Consensus System. *arXiv preprint arXiv:1805.04548* (2018).

[26] Richard Karp, Christian Schindelhauer, Scott Shenker, and Berthold Vocking. 2000. Randomized rumor spreading. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 565–574.

[27] A-M Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. 2003. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed systems* 14, 3 (2003), 248–258.

[28] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. 2017. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*. Springer, 357–388.

[29] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing bitcoin security and performance with strong consistency via collective signing. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 279–296.

[30] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 583–598.

[31] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, Vol. 41. ACM, 45–58.

[32] Klaus Kursawe and Victor Shoup. 2005. Optimistic asynchronous atomic broadcast. In *International Colloquium on Automata, Languages, and Programming*. Springer, 204–215.

[33] Jae Kwon. 2014. Tendermint: Consensus without mining. *Draft v. 0.6, fall* (2014).

[34] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29.

[35] Harry C. Li, Allen Clement, Edmund L. Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. 2006. BAR Gossip. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, Berkeley, CA, USA, 191–204. http://dl.acm.org/citation.cfm?id=1298455.1298474

[36] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. 2016. {XFT}: Practical fault tolerance beyond crashes. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 485–500.

[37] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. 2019. Fast and secure global payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 80–96.

[38] Ben Lynn. 2007. On the implementation of pairing-based cryptography. *The Department of Computer Science and the Committee on Graduate Studies of Stanford University* (2007).

[39] Silvio Micali, Michael Rabin, and Salil Vadhan. 1999. Verifiable random functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 120–130.

[40] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 31–42.

[41] Zarko Milosevic, Martin Biely, and André Schiper. 2013. Bounded delay in Byzantine-tolerant state machine replication. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*. IEEE, 61–70.

[42] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system.

[43] Rafael Pass and Elaine Shi. 2017. Hybrid consensus: Efficient consensus in the permissionless model. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 91. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[44] Rafael Pass and Elaine Shi. 2018. Thunderella: Blockchains with optimistic instant confirmation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 3–33.

[45] Li Peilun, Wang Guosai, Chen Xiaoqi, and Xu Wei. 2018. Gosig: Scalable Byzantine Consensus on Adversarial Wide Area Network for Blockchains. *arXiv:1802.01315* (2018).

[46] Barbara Simons. 1990. An overview of clock synchronization. In *Fault-Tolerant Distributed Computing*. Springer, 84–96.

[47] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolić. 2019. Mir-bft: High-throughput BFT for blockchains. *arXiv preprint arXiv:1906.05552* (2019).

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

[49] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. 2016. Keeping authorities "honest or bust" with decentralized witness cosigning. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 526–545.

[50] Douglas B. Terry, Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, and Daniel C. Swinehart. 1988. Epidemic Algorithms for Replicated Database Maintenance. *Acm Sigops Operating Systems Review* 22, 1 (1988), 8–32.

[51] Florian Tschorsch and Björn Scheuermann. 2016. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Communications Surveys & Tutorials* 18, 3 (2016), 2084–2123.

[52] Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul Francis. 2006. Chunkyspread: Heterogeneous unstructured tree-based peer-to-peer multicast. In *Network Protocols, 2006. ICNP'06. Proceedings of the 2006 14th IEEE International Conference on*. IEEE, 2–11.

[53] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2009. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on*. IEEE, 135–144.

[54] Marko Vukolić. 2015. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *International Workshop on Open Problems in Network Security*. Springer, 112–125.

[55] WonderNetwork. Apr. 2018. Global ping statistics: Ping times between WonderNetwork servers. https://wondernetwork.com/pings.

[56] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014).

[57] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.

[58] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 931–948.

[59] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, and Huaimin Wang. 2017. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services* (2017).