Graph Learning BFT: A Design of Consensus System for Distributed Ledgers

MYOUNGWON OH, SUJIN HA, JIN HYUK YOON, KANG-WON LEE, (Senior Member, IEEE), YONGSEOK SON, AND HEON YOUNG YEOM, (Member, IEEE)

1Department of Computer Science and Engineering, Seoul National University, Seoul 151-742, South Korea
2SK Telecom, Seoul 07899, South Korea
3School of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea

Corresponding author: Yongseok Son (sysganda@cau.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIT) under Grant 2018R1C1B5085640 and Grant 2015M3C4A7065645, and in part by the Korea Institute for Advancement of Technology (KIAT) Grant funded by the Korean Government (MOTIE) under Grant P0012724 (The Competency Development Program for Industry Specialist).

ABSTRACT Distributed ledger technology faces scalability problems due to a long commit time despite recent successes for cryptocurrency. Small group consensus studies have improved this scalability of distributed ledgers. However, they still have problems of the consensus process itself. For example, most blockchain systems perform serialized block proposal and consensus processing, guarantee the finality with high overhead, and handle byzantine nodes inefficiently. To address these problems, we propose a consensus system, named graph learning byzantine fault tolerance (GL BFT), which offers high parallelism and low latency under Byzantine fault. To do this, we enable a parallel pipelined agreement by separating the block proposal and the consensus process. Second, we devise two techniques of merging blocks and commit learning to guarantee the finality with little overhead. Finally, we present a path learning approach which chooses optimal paths to handle Byzantine fault. The proposed GL BFT can achieve instant finality with low message overhead among a small group of nodes even if Byzantine nodes exit. Also, we evaluate its performance on an open source blockchain protocol. Experimental results show that our design reduces data traffic required by the consensus up to 30%, one transaction is finalized within a few seconds, and optimal performance is maintained.

INDEX TERMS Distributed system, blockchain, consensus system.

I. INTRODUCTION

Distributed ledgers, also called blockchain enabled decentralized services, e.g. Bitcoin [1], Ethereum [2], and Hyperledger [3], have been applied to various fields, such as micro payment, cryptocurrency, logistics, Internet of things (IoT), etc. Although blockchains provide high security and transaction processing as a data management system, there are some limitations, including the blockchain consensus protocol has low throughput [4], no guarantee on consistency [5], and lack of scalability [6], [7] compared with traditional centralized systems. Furthermore, consensus time increases with increasing blockchain participants [8]. Thus, there are few practical blockchain implementations, in contrast to perceived prospects that blockchain could provide a new internet platform [9]–[11].

Many recent studies have tried to improve blockchain performance and scalability by selecting node subsets to participate in the agreement [12]. Small random committee approaches [6], [13]–[15] based on verifiable random function (VRF) [16] or hybrid approaches combining proof of work (PoW) and Byzantine fault tolerance (BFT) [5], [17] use small groups to represent the whole group and achieve fast consensus. Similarly, private blockchains delegate a small group of authorized members to represent all members, solving the performance problem by only allowing the small group to participate in consensus rather than open membership [3], [18]. The present study follows these approaches
in terms of using a small group to improve performance and scalability. However, several problems remain as follows.

A. SERIALIZED BLOCK PROPOSAL AND CONSENSUS PROCESSING
Blockchain performance is affected by how many blocks are propagated and which data structure is used by the ledger. Most blockchain schemes employ a few predetermined nodes to propagate blocks and execute block proposal and consensus processes, which significantly reduce performance and scalability. Performance is limited even if many nodes can propose blocks and reach consensus simultaneously due to the blockchain data structure. Most blockchain platforms use single linked list [1], [2] or directed acyclic graph (DAG) data structures [19], [20]. Single linked list approaches reduce performance and scalability, and DAG approaches can be difficult to guarantee the time to achieve finality.

B. FINALITY WITH HIGH OVERHEAD
Blockchain systems can be classified based on how they propagate data to reach consensus, i.e., Gossip [1], [6], [19] or state machine replication (BFT SMR) [7], [12], [22] protocols. Gossip protocol is relatively easy to implement, but cannot guarantee finality; whereas BFT SMR protocol can guarantee finality but requires many messages to reach agreement.

C. INEFFECTIVE HANDLING OF BYZANTINE NODES
In contrast with conventional database systems, blockchains allow malicious nodes to join the network while generating and storing blocks. Therefore, it is essential to address this Byzantine fault for system reliability. Previous studies [12], [22] have tried to solve Byzantine faults through a new phase by re-electing a leader [12] or periodically replacing a leader [22]. However, these approaches allow a new target to be attacked by the malicious nodes, and performance is reduced due to re-election and leader change processes.

Therefore, this paper proposes a consensus system based on a small group to improve scalability and reduces consensus time. The proposed approach can be summarized as follows.

1) We explain the bottleneck of the blockchain system when multiple nodes try to reach consensus and reveal that the main cause is heavy message overhead and serialized data structure.
2) We propose a parallel pipelined proposer comprising multiple proposers and scatter-gather DAG to enable parallel pipelined agreement by separating the block proposal and consensus processing.
3) We propose techniques for merging blocks and learning commit based on node traversal to guarantee instant finality with low overhead.
4) We propose a path learning approach to select optimal network paths to handle Byzantine nodes and hence maintain optimal performance.
5) We show the effectiveness of our design by implementing it in EOS. Extensive evaluations show that our prototype outperforms the original EOS by up to 30% in terms of transactions per second.

We implement the proposed design on open source blockchain, EOS [18]. Experiment results show that our propose system reduces network traffic by up to 30%, reaches consensus in 0.5 seconds in the case of ten validators, and handles Byzantine fault without performance degradation.

The remainder of this paper is organized as follows. Section 2 describes relevant background and previous studies, and Section 3 defines problems with current systems. Section 4 explains relevant assumptions and Section 5 presents the proposed system. Section 6 explains GL BFT correctness and Section 7 presents various experimental results. Section 8 discusses relevant future work and Section 9 summarizes and concludes the paper.

II. BACKGROUND AND RELATED WORKS
A. PROOF OF WORK
Bitcoin [1] or Ethereum [2], which is the most popular proof of work (PoW) based cryptocurrency, generate blocks using hash and link the blocks in a single linked list. If a certain number of blocks are linked, then the block is regarded as confirmed. However, although this enables anyone to generate blocks without a complicated protocol, it leads to the following problems [5].

1) PoW is designed to be independent from any specific node, but a strong miner pool is currently generating most of the blocks. Thus, blockchains can be greatly affected depending on specific nodes having a strong power, which causes a serious security problem.
2) Block generation requires considerable hash calculations, with consequential tremendous power consumption.
3) It is not possible to know when a transaction will be committed, due to the fork [23], which causes various security problems and decreases throughput.

B. BFT PROTOCOLS
Applying the Byzantine Fault Tolerance (BFT) algorithm to the blockchain can solve the problem that PoW cannot guarantee block finality. Practical byzantine fault tolerance (PBFT) [12] that assumes a weak synchronous network model was the first practical protocol to be applied to a real system. However, complicated view changes occur when the leader is the faulty node, with many message exchanges between nodes.

Hot-stuff [7] reduces the number of message exchanges between validators during the leader replacement, and accelerates consensus by ensuring the next block proposal round is executed even if a block is not committed. If descendant blocks are committed, then the parent block is considered as committed. However, Hot-stuff has several limitations.

1) It does not guarantee instant block finality, e.g. the chain Hot-stuff maintains can be divided into several candidate chains, which require synchronization, hence Hot-stuff does not know how long it will take.
2) Although Hot-stuff reduces fork possibility, the possibility is not zero.

Tendermint [22] protocol combines PBFT and delegated proof of stake (DPoS), using a locking mechanism to freeze stakes when voting (hence avoiding double spending) and preventing attacks by removing all the stakes from the malicious node. However, DPoS has heavy leader election protocol if a re-election is required.

C. SMALL COMMITTEE CONSENSUS

Although BFT consensus solves blockchain finality, the scalability problem remains: requiring increasing block consensus time with increasing number of participating nodes. Small committee consensus can solve scalability problems by choosing a small group that participates.

1) DPoS

EOS [18] uses DPoS to select a small number of committee members and propose blocks among them. By choosing the small members which have a lot of stakes, it overcomes the scalability problem. However, EOS takes a long time to commit the block because validators need to propose a block in a round robin manner to be able to vote for the block.

2) RANDOM SMALL COMMITTEE

A direct election of committee members by users takes considerable time, and a powerful group could possibly seize the committee to control the entire network. Previous studies have proposed several approaches based on random election algorithms to solve this issue. Algorand [6] prevents attacks by malicious users via random selection when selecting committee members. Hence it requires less time than all nodes based consensus since there are only a few committee members. However, Algorand does not guarantee block finality, and should some problem occur within a round, e.g. network delay or malicious attack, the round must be re-assessed several times. The block proposal is rejected only when all consensus attempts fail. Therefore, the user does not know whether the transaction is finalized or when the transaction will be applied.

Avalanche [14] was designed to achieve consensus eventually for all nodes by repeatedly choosing a random small group. It maintains two counters to achieve a final consensus: how many small groups agree on the transaction, and how many transactions that reach the consensus are consecutive. However, Avalanche cannot know when the transaction is finalized, since finality is from multiple consensus between small groups.

D. DIRECTED ACYCLIC GRAPH (DAG)

Blockchains generally use a single linked list as the main data structure, but this can reduce performance and scalability since it cannot reference multiple blocks simultaneously. Therefore, DAG was introduced since to link multiple blocks.

SPECTRE [20] uses similar block proposal to PoW and reaches consensus by allowing all blocks to vote on a block pair through the DAG structure. After a DAG is created at the end of the block proposal, voting commences to agree on the transactions in DAG. However, linking blocks to the DAG does not guarantee that the blocks are committed since the block sequence among the DAG generated at the block proposal is only finalized when voting completes. Therefore, the DAG is not a finalized sequence even when it is generated.

IOTA [19] uses a DAG structure called tangle. To add a transaction to the tangle, a node must validate two transactions in the blockchain network. If the node confirms two transactions, it can add a new transaction that references the previous two transactions. Therefore, all DAG transactions are not committed, and only become committed only after the node adds a new transaction to verify it. Thus, IOTA does not guarantee instant finality.

E. DISTRIBUTED LEDGER

The distributed ledger is the technology that makes all of the participant nodes to share the synchronized data across multi-sites without a central authority. Before updating the data in the ledger, each participant node needs to reach an agreement. Therefore, the consensus algorithm between nodes is crucial in the distributed ledger.

Blockchain is one form of the distributed ledger. Blockchain manages ledgers which is spread across multi-sites through peer to peer networks, and it maintains its state consistently by using consensus algorithm, called PoW. Also, it appends data as an append-only database.

Bitcoin [1] and Ethereum [2] are well-known blockchain platform that uses gossip protocol and PoW to reach a consensus. Unlike Bitcoin, Ethereum features Ethereum virtual machine (EVM), allowing the smart contract to execute on the ledger.

Hyperledger fabric [3] is the most popular private blockchain platform. It proposed execute-order-validate architecture instead of conventional order-executed architecture which is widely used. By decoupling execute stage, each transaction can be submitted to endorsers in parallel, and the client sends the ordered transaction if there is no conflict. Hyperledger fabric also maintains ledgers by using an append-only data structure. In addition, a smart contract, called chaincode, runs during the execution phase.

Multichain [21] is a private distributed ledger platform. Although it derived from Bitcoin, multichain makes use of other approaches to solve problems such as mining and permission. To achieve fairness during mining, multichain uses electing block proposed in a round-robin manner. Also, every node in multichain manages the permission list to involve trust nodes to the network.

III. PROBLEM DEFINITION

A. LOW PERFORMANCE AND SCALABILITY

Figure 1 shows the consensus process for modern blockchain systems comprises two different methods: With certificate and No certificate. No certificate can prove majority vote for the proposal without certificate [1], [2], e.g. the block is
confirmed under PoW if there are some threshold number of blocks behind the base block. In contrast, With certificate validators leave evidence that they have voted [22]. For modern blockchain systems, each block contains the certificate for the previous block, and a block is considered valid if two or more blocks are normally generated. However, these methods have the following problems.

1) SINGLE BLOCK PROPOSER
Current BFT protocols [7], [12], [24], [25] have a single proposer (leader) that can make a proposal request. Thus, more than one consensus cannot be reached even though many nodes participate in the blockchain network. A multiple block proposer is required, generating multiple proposal requests, to efficiently reach consensus when many nodes participate.

2) DATA STRUCTURE LIMITATIONS
Most modern blockchain systems use the single linked list data structure, where only one block can be linked to the list at a time since each block requires a unique reference to the previous block. Thus, the list cannot reference multiple blocks if many block proposers issue blocks simultaneously. DAG data structures can solve single linked list disadvantages by concatenating multiple blocks in parallel.

3) SERIALIZED BLOCK PROPOSAL AND CONSENSUS PROCESSING
Most blockchain systems have tightly coupled block proposal and consensus processing, e.g. a certificate must be acquired or information about the previous valid block must be available to propose the next block. This can reduce performance and scalability by serializing block proposal and consensus processing.

B. GUARANTEEING FINALITY
Instant finality is essential for blockchain systems that require reliable transaction processing since systems with guaranteed finality have several critical advantages.

1) Facilitating data exchange and interworking. The mechanism makes those operations easy to connect an inter-chain [25] or side chain [26] and enable Atomic Swap [27]. If finality is not guaranteed in the inter-chain [25], then actual implementation will be difficult. Since it is not possible to know when data transaction processing is completed, data exchange will become uncertain.

2) The client can make the next transaction easier. Without finality on the blockchain system, the client does not know which transaction need to be issued; whereas client transaction processing speed will be faster if instant finality is achieved.

3) The fork can be removed. As mentioned in Section II, if there is fork, security and performance problems arise.

1) LACK OF FINALITY
Bitcoin [1] and Ethereum [2] build distributed ledgers to enable all participants to reach a consensus. However, clients cannot know when a transaction is committed due to forks and gossip protocol limitations. Most approaches [18]–[20] have this problem, and hence cannot guarantee instant finality.

2) TRAFFIC OVERHEAD
PBFT based approaches [5], [22], [28] can achieve finality and reach a consensus quickly within a small number of nodes. But various performance issues remain. For example, PBFT requires three communication phases to address Byzantine faults, which increases network traffic and has high message complexity (O(n^2)). Quality control (QC) chain protocol [7] message complexity can be reduced to reach a consensus (O(n)), but requires message broadcasts for block proposals and collecting certificates over the three phases.

C. INEFFICIENT FAULT HANDLING

1) FAULT RECOVERY
Even if a fault occurs on a participating node (i.e., the node is Byzantine), there can be no problem under PoW when there is sufficient nodes in the blockchain because the current nodes will not consent to including the Byzantine node. However, if the Byzantine node is given the leader role, the role and configuration for each node must be changed depending on which node is faulty. For example, if a failure occurs in PBFT on the leader node, a new leader must be elected via view-change phase. This can significantly increase protocol and message complexity and cause security problems [29], [30].

2) LACK OF ADAPTATION TO NETWORK CHANGE
Previous studies have addressed BFT solutions under fixed network conditions (e.g. limited number of nodes). However, the optimal path between nodes for blockchain networks can change continuously change since some nodes can Byzantine or under heavy load. Thus, an adaption solution to network change is required for blockchain systems.

IV. ASSUMPTION
The proposed solution assumes a weak synchronous network model. We set the time bound so that the client receives
a response within a finite time to provide liveness. Liveness and safety are discussed in Section 5. We use asymmetric cryptography. All propagated messages and blocks contain the sender’s digital signature. We do not mention how nodes join the network. It is assumed that nodes in the network already know each other, and all nodes know the public key of another node. We standardize notations to be used in this paper. \( n \) is the total number of nodes, \( q \) is the number of correct nodes and \( f \) is the number of fault nodes. We also assume that the fault bound is \( f = \lceil (n - 1) / 3 \rceil \), which is the same as PBFT [12].

V. DESIGN AND IMPLEMENTATION

Figure 2 shows the proposed GL BFT system to solve the problems discussed above, comprising three phases: parallel pipeline, merging blocks/learning-commit, and path learning; where each phase is described in more detail in the subsequent sections.

A. PARALLEL PIPELINED PROPOSER

Figure 3 shows the proposed parallel proposer pipeline with two schemes: multiple proposers and scatter-gather DAG, to solve performance scalability described in Section III-A. Each validator can propose a block simultaneously, and these blocks can also be stored simultaneously. Thus, all validators can generate blocks and request consensus. This processing decouples the consensus process from block proposal, and hence consensus and block propagation can be performed in parallel. Also, this parallel pipelined propose phase performs the block propagation and the validation.

Once parallel pipelined propose phase is completed, the validator that completes the block proposal assumes that other validators have the proposed block. Thus, any validator can start the consensus on a proposed block. The parallel pipelined propose message has the form (parallel pipelined propose, block id, epoch, base block id, block). With the parallel pipelined propose message, the validator recognizes the base block (base block id) and block content (block). Parallel pipelined propose phase is comprised of two parts which are multiple proposer and scatter-gather DAG as follow.

1) MULTIPLE PROPOSERS

We propose multiple proposers in the parallel proposer pipeline to enable multiple proposals. The client signs a transaction with his private key and propagates it to the blockchain network. If the validator that received the transaction can determine the transaction is valid, the transaction is included in the next block generated by the validator. The generated block is passed to other validators with base block information and a consensus request for this block is made after a given time. If validators fail to reach consensus, the process repeats from the first step. The proposed approach is similar to the Hyperledger fabric execution-order-validation design [3]. However, validators are responsible for all roles rather than having a separate endorser or orderer.

Figure 4 shows the proposed multiple proposer procedure. When a validator receives transactions from clients, it generates a block that includes valid transactions based on its base block, and then broadcasts the block to other validators. Each validator makes a consensus request for blocks it has at regular intervals. Multiple proposers enables simultaneous request blocks for consensus. However, This is not sufficient to achieve higher performance and scalability. For example, the single linked list data structure cannot enable high parallel operations and block proposal and consensus processing remain serialized.

2) SCATTER-GATHER DAG

We propose scatter-gather DAG rather than current single linked list data structure to achieve higher performance and scalability. Figure 5 compares consensus processing for single linked list and scatter-gather DAG data structures.
In contrast with linked list based approaches (Figure 5-(a)), blocks can be added and linked via scatter-gather DAG (Figure 5-(b)).

With this scheme, finality can be immediately achieved by adding certificates which indicate that two-thirds of the entire validators in the network agree on that block. Thus, the proposal process and the certificate process can be separated, which makes the entire agreement procedure pipelined. Scatter-gather DAG can continue to expand if there is no dependency between the blocks. However, scatter-gather DAG acts as a single linked list if the blocks are related; and conflict can occur between transactions, reducing performance.

Algorithm 1 Proposed Parallel Proposer Pipeline Procedure

| Input: user_transactions |
|--------------------------|
| Output: status_code      |

1. foreach transaction in user_transactions do
   2. if transaction has conflict to read_write set at epoch then
      3. return ResponseConflict;
   4. if check transaction is invalid then
      5. return ResponseInvalid;
   6. ProduceBlock();
   7. BroadcastBlocks();
   8. if Time delay expired then
      9. SetTimeOut();
     10. DoMergePrepare();
   11. return ProducedNotConfirmed;

### a: HANDLING CONFLICTS

To handle a conflict between blocks, validators manage read and write sets at each epoch (Algorithm 1, line 2). Validators use these sets to determine if a block can be attached at the epoch. Figure 6 shows how conflicts occur and how they are handled when a block is proposed.

- Case 1. Validators agree on block B because there is no block in the next epoch of block A.
- Case 2. Since validators already have block C in the current epoch, they check block B’s read and write sets, and then agree on block B only if there is no conflict with block C’s read and write set.
- Case 3. Block E is in the next epoch. However, if block B does not conflict with all blocks in the current and previous epoch, it can be added as a leaf node permanently that does not link with block E. Subsequent new block proposals should omit referencing this leaf node block.

### b: BLOCK PROPOSAL AND CONSENSUS SEPARATION

Validators propose the block based on its own Scatter-gather DAG (Algorithm 1, line 7), and can request a consensus for the block independently of the block proposal (Algorithm 1, line 10). This is possible because several blocks in an epoch can be concurrently linked to a Scatter-gather DAG and validators can request a block concurrently without the leader.

### B. MERGING BLOCKS AND LEARNING COMMIT BASED ON NODE TRAVERSAL

We propose merging block and learning commit techniques based on node traversal [31], [32] to provide instant finality with low overhead. A consensus request can be made by any validator in the network. Once the consensus request is created, it is passed through the optimal path from the traversal map that the validator has. This request is not propagated to all nodes simultaneously, but propagates individually in the order they are listed in the map.

Node traversal method can reduce the message overhead to $O(n)$. This is better than the existing PBFT-based agreements which require $O(n^2)$ [12], [22] and similar to Hot-stuff [7], but the actual number of messages is better than it (GL BFT needs $3n$ messages and Hot-stuff needs $8n$ messages). Message throughput is closely related to computing power and network traffic load, which is a prerequisite for ensuring instant finality. Although latency may increase due to node traversal, overall consensus time can be reduced because the consensus process can commence while the block proposal is being.
processed (Section V-A). If the network is congested, high message complexity will also significantly increase overhead.

1) MERGING BLOCKS BASED ON NODE TRAVERSAL
The Merge-prepare step forwards the consensus request to the selected path. At the beginning of this phase, each validator that has completed the parallel proposer pipeline requests a merge-prepare message after some time delay. The time delay minimizes potential contention between different validators because they can issue merge-prepare request for the same block. The Merge-prepare message has the form (merge-prepare, block id, epoch, base block id, H(traversal map entry), signatures), and the merge-process procedure is as follows.

1) The first validator creates merge-prepare message with the traversal map entry. At this time, the validator learned the traversal map entry. For example, assuming that there are 4 validators and the id of each validator is 1, 2, 3 and 4, the path such as 1-3-2-4 can be the entry of the traversal map. Another validator that receives H (traversal map entry) will find the traversal map entry in its own entries. The validator considers the request as fail if H (traversal map entry) does not exist, and makes no response.

2) When the merge-prepare request is received, the validator adds the signature, and then forwards the merge-prepare request to the next validator (Algorithm 2, line 18). If the validator does not agree, it does nothing.

3) If two-third of the entire validators agree on the block, Learning-commit is started (Algorithm 2, line 20), and the remaining one-third validators will receive the merge-prepare message asynchronously (Algorithm 2, line 23). One validator can propose only one block per epoch. After a period of time, the epoch moves on to the next epoch. Each validator keeps its merge-prepare message until timeout, and if a new request conflicts with the current processing request, it discards the new one.

2) LEARNING COMMIT BASED ON NODE TRAVERSAL
This phase propagates the message in reverse order of the merge-prepare traversal path. When the validator receives the learning-commit message, it determines if the majority of validators agree to the merge-prepare request, and if so, commits the block (Algorithm 3, line 5) and sends the learning-commit message to the next validator. The validator also updates its own traversal map entry considering the merge-prepare cost (Algorithm 3, line 6), and this updated information is used in the parallel proposer pipeline for the next epoch. Cost can be measured in various ways; the present study used latency.

The final validator in the learning-commit phase is the validator that triggered the first consensus request. Consensus is completed when the learning-commit message arrives at this validator, and commit response is returned to the client.

```
Algorithm 2 Merging Blocks Procedure
Input: merge_prepare
Output: status_code
1 block_id = merge_prepare.block_id;
2 block = getBlock(block_id);
3 if in_progress_merge_prepare has merge_prepare then
4 // Time to handle merge_prepare expired
5 if Timeout then
6 Enqueue merge_prepare to in_progress_merge_prepare return Done;
7 else
8 return AlreadyProgress;
9 if !VerifySignMergePrepare() then
10 return InvalidRequest;
11 foreach transaction in block.transactions do
12 epoch = block.epoch;
13 if transaction has conflict with read_write set at epoch then
14 return ResponseConflict;
15 if check transaction is invalid then
16 return ResponseInvalid;
17 // agree with merge_prepare, so send next validator this merge_prepare
18 SendMergePrepare(merge_prepare);
19 if merge_prepare.vote_count == enough votes then
20 SendLearningCommit();
21 MakeBlockFinalized();
22 else if merge_prepare.vote_count > enough votes then
23 MakeBlockFinalized();
24 return Done;
```

```
Algorithm 3 Procedure of Learning-Commit
Input: learning_commit
Output: status_code
1 block_id = learning_commit.block_id;
2 block = getBlock(block_id);
3 if !VerifySignLearningCommit() then
4 return InvalidRequest;
5 MakeBlockFinalized();
6 UpdateTraversalPath(learning_commit);
7 if this node the first entry of learning_commit.traversal_path then
8 return Done;
9 SendLearningCommit(learning_commit);
10 return Done;
```

C. PATH LEARNING
Typical blockchain networks do not guarantee low latency or high security, and any node can become slow or faulty even if a small committee joins an agreement process.
Therefore, we need to consistently find the best path to reach agreement and hence ensure reliability and maintain optimal performance. Thus, we combine continuous learning concept with BFT and add a design that learns optimal and candidate paths. Path learning is independent of the GL BFT process, operating periodically to learn the latest network state and find the best path to reach consensus. There are three states when each validator performs path learning.

1) Gathering. Collecting path information from other nodes.
2) Learning. This state has two jobs:
   a) Set priorities for learned paths which have path information from other nodes. The validator gets this information from the self-learning or learning-commit phase, and the cost (i.e., latency) for each path is considered when determining the optimal path priority.
   b) Register a candidate path and query the state of that path to the network. If the path is valid, the validator receives a request containing cost information ((learning-request, traversal-map-entry, cost)).
3) Proposal. The validator broadcasts its path to other validators ((learning-propose, traversal-map-entry, cost)).

The learning state restarts when the proposal state completes, repeating the three states.

1) TRIGGER TO CHOOSE A CANDIDATE PATH
Each validator changes the optimal path in two cases.
   1) No response until certain time interval (less than timeout to fail) when the merge-prepare phase is started.
   2) Change the existing path to optimal path periodically among the currently learned paths.

2) SYNCHRONIZATION
The proposed protocol incorporates two synchronization types.
1) One validator is responsible for collecting finalized blocks during one epoch to make full-DAG (DAG includes all finalized block in one epoch to ensure protocol safety). As discussed above, each node can propose one block for an epoch. After proposing a block, the node must wait wait time for the next epoch before proposing another block, where wait time is the time between epochs. The node is responsible for synchronization changes for each epoch in a round robin manner. It requests the latest finalized blocks in the current epoch, checks their certificate, and combines all received blocks with valid certificated into a full-DAG.
2) Full synchronization, which ensures that all correct nodes have a same DAG. This is run periodically by user configuration.

VI. FAULT HANDLING
A. MERGING BLOCKS PHASE
If the merge-prepare phase cannot be finished due to a malicious node or exception, this error will be managed as the following three cases after a certain time delay.
1) The validator that originally sent the merge-prepare request resends a request message again with a different path.
2) Another validator resends the merge-prepare request for the block.
3) The validator discards the consensus request if the requested epoch is considerably later than the current epoch because the majority of validators do not agree on that request.

An injection attack during the merge-prepare phase is impossible because a signature of the validators in the traversal map is required to reach agreement.

B. LEARNING-COMMIT PHASE
A fault occurs when validators have agreed to accept the block, but a learning-commit request has not yet been propagated. The way to overcome this fault is that another validator reissues the same request. For example, suppose a fault occurred for the first learning-commit validator. Then the requested block will be requested again after a given delay because it has already been propagated to the validators. If the validator receives a request that conflicts with the current block, it discards the incoming request, preserving the previous merge-prepare message it agreed to.

Faults occurring on the second or third validators in the learning-commit phase are recoverable in the same manner.

C. REPEATED REQUESTS FOR IDENTICAL TRANSACTIONS
Suppose a malicious client sent the same transactions to all validators. We can address this problem as follows. For correct transactions with the same content, at least a single block that contains correct transactions is finalized and other blocks are rejected. Therefore, validators create the next blocks without the transactions because they create the next blocks based on the current DAG status. Although this could impact on performance, it will not affect safety.

We can avoid this attack if we add a black-list policy that prevents attacks, but we leave this scheme for future study.

In the case of incorrect transactions with the same content, these transactions will be rejected by correct validators before starting merge-prepare phase.

VII. CERTIFICATE MANAGEMENT
We propose a common certificate method to check that validators have agreed on the block. Merge-prepare and learning-commit messages contain a signature to prove that each validator has agreed on the block, recorded in the same way as Tendermint [22], [25]. Thus, the number of certificates and total size increase as the number of validators increases, which will become a serious problem for extremely
large networks. In principle, threshold signatures [33] or zero knowledge proof [34] can be used to address this challenge, but we have left this aspect for future study.

VIII. CORRECTNESS
In this section, we use the term “nodes” that indicates validators and block proposers.

A. SAFETY
The proposed GL BFT framework incorporates scatter-gather DAG which is the total order of blocks. Therefore, safety can be proven by showing that correct nodes have the same DAG with the same block order. To show this proposition is true, we first prove there are no blocks that conflict in the DAG; then show that different DAGs can be merged in an epoch; and finally prove that it will eventually update the DAG, even if the correct node has the old DAG for a while.

**Theorem 1:** There are no blocks that conflict each other in the DAG.

**Proof:** In merge-prepare phase, if the correct node has received the proposed block, it checks the read and write set to confirm whether the proposed block has conflict with other blocks being processed in the current epoch. The block will not be able to obtain sufficient votes if it is invalid and will be discarded.

**Theorem 2:** If node A and node B have different DAG-A and DAG-B in epoch e, the two DAGs can be merged.

**Proof:** Suppose that block A attached to DAG-A and block B attached to DAG-B, but block B has conflict with block A. From Theorem 1, if block A is finalized, block B cannot be finalized. Therefore, the blocks can be merged.

**Theorem 3:** Even if the correct node has the old DAG during epoch e, it will have the latest DAG eventually.

**Proof:**
- Network delay: If the communication is delayed for some time, even the correct node may not be able to update the DAG. However, since we assume a weak synchronous network model, the correct node will have the latest DAG after a certain time bound.
- The committed blocks are not propagated due to malicious nodes: Suppose the traverse map has a malicious node B and correct node A next to B. Node A needs to receive a message from node B to reach consensus, but node B will not deliver a message because node B is malicious node. However, in our protocol, there are two synchronization schemes. With these schemes, the node can combine different DAGs into one by Theorem 2. So, the correct node will have the latest DAG after a certain time bound.

B. LIVENESS
To prove that liveness is guaranteed, we show that the client receives a response within a finite time.

**Lemma 1:** If the system is not broken, there is at least one path to propagate blocks between nodes. If the absence of a path to communicate between nodes, it means that the system or network is dead. This is out of the scope.

**Lemma 2:** If the block is not confirmed after a certain time, it will retry the consensus process through another path. As described in Section VI, validator reissues the same request for overcoming the fault.

**Corollary:** Client receives a response a finite time.

In the normal case, GLBFT will find the optimal path through path learning, quickly agree on the block through that path, and then reply to the client. In the worst case, GLBFT will give the response to the client after n! retries. If the normal path is in the last order of path candidates, the block will reach consensus after going through a retry of n!, the number of all paths. Therefore, the block containing the client’s transaction can reach an agreement in a finite time and the client can receive a response. In addition, failure probability continues to reduce with increasing number of retries, and since the nodes learn the path continuously, the probability the block is for committed increases.

IX. EVALUATION

A. ENVIRONMENT SETUP
We implemented the proposed design on EOS 1.3.1, employing EOS clusters comprising ten server nodes; each with Intel Xeon E5-2690 2.6Ghz (12 cores), 128 GB RAM and SSD (Intel 512GB). Eight client nodes were connected to the cluster through 10GbE NIC. Each server ran EOS daemons configured with base parameters and each EOS ran on the XFS as a local file system. Our evaluation was based on EOS’s real service, and clients requested transactions through Cleos CLI. Cleos sent HTTP requests to Nodesos to manage transaction requests.

B. FINALITY TIME
Figure 7 shows load testing for ten validators with respect to the number of clients. Each client issued approximately 100 transactions per second (TPS). The transaction structure we evaluated was the same as the EOS transaction structure. Latency was measured by the client when the block was finalized, i.e., latency was the finalized time from the client point of view. Cases (a), (b), (c), and (d) tested the ideal case, i.e., transactions were all independent. The validator generated the block at fixed period (0.5 ms) and the result for a single block producer was based on a single linked list. In contrast, the result for multiple block producers was based on scatter-gather DAG.

Figure 7-(a) shows current scheme performance for a single block producer. TPS is increased up to 600 for low load, but reduced (i.e., latency increased) as the load increased beyond seven clients. Figure 7-(b) shows TPS change for the current scheme with eight clients. Since one producer creates a block, TPS is not constant under heavy load.

Figure 7-(c) shows the proposed scheme performance for ten block proposers, i.e., each server node has a block proposer. TPS increased to approximately 800 as the number of clients increased. Figure 7-(a) shows that average latency increased from approximately 2.5 s and plateaued.
at 0.38 s (Fig. 7-(c)). Maximum and minimum latency also remained relatively constant, since if only one validator was used, the probability that a transaction would be included when a block was generated reduced, hence the time to finalize a transaction increased. On the other hand, when multiple validators issue blocks simultaneously, the latency for all test cases are similar even if the number of clients increases (Figure 7-(c)), because all validators can generate the block and attempt to request consensus for a block. Figure 7-(d) confirms that multiple producers provide constant TPS.

**Worst Case:** Figure 7-(e) shows the result of issuing conflicted transactions by each client when the same experiment (the result of eight clients of Figure 7-(c)) was conducted. Each client performed a transfer operation on two arbitrary accounts among 10 accounts, they requested 100 TPS, and the response of transaction fail due to the conflict is excluded from the experimental results. The number of completed transactions is low due to conflicts between transactions. However, the latency of finalized transactions is slightly increased compared with the ideal case of Figure 7-(c) (ideal case: 0.32 s, worst case: 0.54 s). This result shows that we have isolated the consensus and the block proposal, so the conflict do not affect the latency of the consensus process. As a result, even if a lot of conflicts occur, only TPS is decreased.

Table 1 shows the result of latency analysis for each of the sub-actions when ten validators produce blocks as shown in Figure 7-(c). We measured sub-actions taken by the validator (i.e., transaction processing (TxP), block generation and propagation (BlockP), consensus processing (ConsensusP), transaction post-processing (TxPP)) when the client issues transactions. The transaction processing (e.g., check validation and conflict) and the transaction post-processing (e.g., ack to client) have no significant effect on overall latency. When the validator finalizes a transaction, it spends most of time to produce and propagate the block. This is because validators produce a block every 0.5 s, if a transaction was not included in the block within this period, then the transaction must wait least 0.5 s. Also, generated blocks need to be propagated to all validator to reach consensus.

The result of Consensus processing requiring node traversal (merge-prepare and learning-commit) shows its latency is relatively low compared with block generation and propagation. This means that block generation and propagation is a major reason of delay to finalize a transaction. The proposed design separates the block proposal and the consensus process so that the consensus process can start during the block proposal is being handled. The average result (0.325 s) is smaller than the sum of the consensus and the block generation and proposal. Thus, even if the number of validators increases and the node traversal takes a long time, the actual latency can be reduced due to pipelined processing.

### C. LATENCY ANALYSIS

| Latency analysis (seconds) | TxP | BlockP | ConsensusP | TxPP | Total |
|----------------------------|-----|--------|------------|------|-------|
| Min                        | 0.002 | 0.033 | 0.025 | 0.002 | 0.045 |
| Max                        | 0.004 | 0.701 | 0.098 | 0.004 | 0.8   |
| Avg                        | 0.003 | 0.305 | 0.045 | 0.003 | 0.325 |

### D. RESOURCE USAGE

Figure 8 shows network throughput and required CPU time to finalize a transaction. We ran the traffic test with increasing number of block producers, and measured EOS, PBFT, and
GL BFT without batching. The methods were implemented on top of EOS. Clients issued transactions at the same rate for all experiments. EOS requires larger data quantities than GL BFT because new blocks must be created and propagated before finalizing transactions in the block. New blocks need to be linked as a chain. Therefore, to obtain attestation from a two-thirds super-majority quorum, a validator must create a block in the specified order. On the other hand, since GL BFT can proceed the consensus separately from the block generation, few resources were required to finalize the transaction.

![FIGURE 9. Faulty node and path learning test.](image)

### E. FAULTY NODE TEST
To evaluate fault handling, we ran experiments and injected faults. Figure 9-(a) shows the result of two faults when clients submitted transactions without conflicts to ten validators. We changed some messages to simulate a fault situation and then forwarded them to the next node. Each client issued 100 TPS. A validator acted as a malicious node at 36 and 72 seconds, respectively. Only slight TPS degradation occurred under the fault conditions. Since each validator has a different optimal path, some changed their optimal path when the fault occurs. Original TPS was recovered some time after the fault occurrence, by learning a new path, and was also recovered even when another fault occurred at 72 seconds. Total TPS was maintained even though there are two faults because the remaining nodes could handle the transactions (i.e., 800 TPS).

We ran the following experiment to demonstrate that GL BFT continues to learn the optimal path. Every validator shares a single optimal path and clients issue transactions at constant rate. This configuration causes a bottleneck because every validator shares a single optimal path. Figure 9-(b) shows the results of path learning. After a certain period of time, a heavy load process that uses almost all of the CPU path was started in the middle of the traversal. Validators can respond to the consensus request under heavy load, but performance is low. We stop the heavy load process at 96 seconds and then restart the process on another node. As shown in the figure, performance was low for a certain interval but then restored due to latency based path reselection when path learning was enabled, as expected; whereas continuous low TPS occurred without the learning process.

### X. LIMITATIONS AND FUTURE WORK

#### A. OVERHEAD OF CONFLICT DETECTION
In the parallel proposer pipeline phase, validators check for conflict between blocks in an epoch to create the scatter-gather DAG. Thus, validators manage the read and write sets in the current epoch and check all read and write sets for new incoming blocks. The conflict detection we used is the same as one in Hyperledger fabric [3]. Since it detects conflict between transactions, if any transaction in conflicts with any other transaction, the blocks have failed to reach consensus and the failed consensus must be tried again, which occurs an overhead.

#### B. TIMEOUT PARAMETER
We rely on a static timeout value. We employ on a static timeout value and Merge-prepare phase commences after the timeout, hence the timeout can affect performance. Future work will establish optimal timeout using CSMA/CA [35].

#### C. DYNAMIC MEMBERSHIP
We used a static small group for the prototype implementation, hence all nodes knew each other’s public key and which nodes were participating in the network. Therefore, we need to consider how to handle key management, current members leaving, and new members joining [36]. The Gossip protocol [6], [37] could be suitable to probe these aspects, and future work will extend the propose GL BFT protocol to cover dynamic membership.

#### D. SCALABILITY TEST
The purpose for the present study was to describe GL BTF small group consensus method for distributed ledgers, and we did not consider large scale tests, i.e., thousands of nodes. However, the proposed approach improved throughput and latency by separating block proposal and consensus, and guarantees finality based on the optimal path, even if latency increases with increased nodes.

#### E. CONSENSUS SCALING
Algorand [6] offers a promising approach to achieve scalability, using a small randomly chosen committee. However, Algorand’s Byzantine agreement protocol does not guarantee finality. The proposed GL BFT approach could be combined with Algorand to provide consensus scaling: select a small committee using Algorand (or a similar VRF based method), and then like and then employ GL BFT to reach consensus. This combined approach is likely to provide finality with low latency, and will be investigated in future study.

#### F. LOAD-AWARE PATH LEARNING
We assumed the global clock was synchronized and measured message latency to find an optimal path. However, some other approaches could measure path latency more accurately [38], [39]. Therefore, we will investigate the advantages of these approaches to optimize the proposed GL BFT solution.
G. Expanding GL BFT

With Zero knowledge proof [34], the size of signatures in each message can be reduced because it proves an agreement between all validators as a single representative signature instead of all validator’s signature. Therefore, using Zero knowledge proof, GL BFT can deliver a message to the next validator with no signature overhead. This optimization can make GL BFT overcome the limitation of membership scalability.

Byzantine fault detection [40] is a key concept to maintain the cluster network to keep stable. If the cluster network can detect Byzantine fault nodes and evict those nodes, a quorum to reach an agreement of the validator can be reduced. As a result, the number of messages required to reach the consensus can be less than before if Byzantine fault detection applies to GL BFT. This is because GL BFT can use the reduced quorum if we keep the number of Byzantine nodes small.

XI. The Effectiveness of the Proposed Approach

Bitcoin [1] and Ethereum [2] are blockchain platforms based on PoW. PoW solves open-membership that anyone can participate in the blockchain network. However, it causes serious performance degradation. To mitigate such a problem, we propose small group consensus, called GL BFT instead of PoW to achieve instant finality and performance.

Tendermint [22] is based on PBFT. GL BFT can run with Tendermint via replacing PBFT with GL BFT. Moreover, GL BFT provides advantages such as low message overhead and maintaining performance compared with PBFT.

DAG-based solutions such as IOTA [19] achieve higher throughput than existing approaches which use a single-list data structure. However, it is hard to guarantee the finality quickly. In GL BFT, with considering consensus protocol and underlying data structure with limited DAG, our proposal can achieve good throughput and the finality.

Hyperledger Fabric [3] is a private blockchain platform, which supports a pluggable interface to change consensus algorithm depending on system environment. Also, since Hyperledger Fabric is based on a small group consensus, GL BFT can easily be applicable.

XII. Conclusion

This paper proposed a consensus system, graph learning Byzantine fault tolerance (GL BFT), to provide high parallelism and low latency under Byzantine fault. We devised parallel pipelined agreement by separating block proposal and consensus processes, and presented merging blocks and commit learning techniques to guarantee finality with little overhead. Finally, we proposed a path learning approach to select optimal paths to bypass Byzantine faults. We implemented the proposed system on a small network and evaluated the performance on an open source blockchain protocol. Experimental results demonstrated that the proposed system reduced data traffic required for consensus by up to 30%, finalizing each transaction within a few seconds, while maintaining optimal performance. Also, we showed GL BFT uses fewer resources than an existing solution, called EOS, to finalize the transaction and demonstrated failure can be recovered faster than original EOS due to path learning.

Acknowledgment

Myoungwon Oh was with SK Telecom, Seoul 07899, South Korea. This work was supported in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIT) under Grant 2018R1C1B5085640 and Grant 2015M3CA7065645, and in part by the Korea Institute for Advancement of Technology (KIAT) Grant funded by the Korean Government (MOTIE) under Grant P0012724 (The Competency Development Program for Industry Specialist).

References

[1] S. Nakamoto. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[2] V. Buterin and Ethereum Foundation. Ethereum White Paper. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper
[3] E. Androulaki et al., “Hyperledger fabric: A distributed operating system for permissioned blockchains,” in Proc. 13th EuroSys Conf., 2018, pp. 30:1–30:15.
[4] I. Eyal, A. E. Gencer, E. G. Sirer, and R. V. Renesse, “Bitcoin-NG: A scalable blockchain protocol,” in Proc. 13th USENIX Symp. Networked Syst. Design Implement., 2016, pp. 45–59.
[5] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, J. Khoffri, L. Gasser, and B. Ford, “Enhancing bitcoin security and performance with strong consistency via collective signing,” in Proc. USENIX Secur. Symp., 2016, pp. 279–296.
[6] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, “Algorand: Scaling Byzantine agreements for cryptocurrencies,” in Proc. 26th Symp. Operating Syst. Princ., Oct. 2017, 51-68.
[7] M. Yin, D. Malkhi, M. K. Reiter, G. Golan Gueta, and I. Abraham, “HotStuff: BFT consensus in the lens of blockchains,” 2018, arXiv:1803.05069. [Online]. Available: http://arxiv.org/abs/1803.05069
[8] J. Abadi and M. Brummermeier. (2018). Blockchain Economics. [Online]. Available: https://scholar.princeton.edu/sites/default/files/markus/files/blockchain_paper_vlog.pdf
[9] Golem. [Online]. Available: https://golem.network/
[10] Gridcoin. [Online]. Available: https://gridcoin.us/
[11] iExec. [Online]. Available: https://iexec.ai
[12] M. Castro and B. Liskov, “Practical Byzantine fault tolerance,” in Proc. 3rd USENIX Symp. Operating Syst. Design Implement., 1999, pp. 173–186.
[13] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, “OmniLedger: A secure, scale-out, decentralized ledger via sharding,” in Proc. IEEE Symp. Secur. Privacy (SP), May 2018, 583-598.
[14] Team Rocket. (2018). Snowflake to Avalanche: A Novel Metastable Consensus Protocol Family for Cryptocurrencies. [Online]. Available: https://ips.is/o/ips/OnU4j3h5mGNzW/Kkijies1RW4YuVj5o2FYqPNPV YwrWGV
[15] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, “The honey badger of BFT protocols,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., Oct. 2016, pp. 31–42.
[16] S. Micali, M. Rabin, and S. Vadhan, “Verifiable random functions,” in Proc. 40th Annu. Symp. Found. Comput. Sci., 1999, pp. 120–130.
[17] R. Pass and E. Shi, “Hybrid consensus: Efficient consensus in the permissionless model,” in Proc. LIPICS-Leibniz Int. Inform. Work., Wadern, Germany: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, vol. 91, 2017.
[18] Esmio. [Online]. Available: https://esios.io
[19] Serguei Popov. (2018). The Tangle. [Online]. Available: https://assets.cfassets.net/1ld6rvzflkhe/2t4uxvbq4k3u0eanh2swog/ 45eae3367ca92f85d94a23a18e1ce1ota1_4_3.pdf
[20] Y. Sompolinsky, Y. Lewenberg, and A. Zohar. (2016). SPECTRE: Serializability of Proof-of-work Events: Confirming Transactions Via Recursive Elections. [Online]. Available: https://eprint.iacr.org/20161159.pdf
[21] MultiChain. [Online]. Available: https://www.multichain.com/
[22] E. Buchanan, “Tendermint: Byzantine fault tolerance in the age of blockchains,” M.S. thesis, Univ. Guelph, Guelph, ON, Canada, Jun. 2016.
[23] A. Baliga, “Understanding blockchain consensus models,” Persistent, vol. 2017, no. 4, pp. 1–14, 2017.
M. Oh et al.: Graph Learning BFT: A Design of Consensus System for Distributed Ledgers