mathematics

Article

Rapid Consensus Structure: Continuous Common Knowledge in Asynchronous Distributed Systems

Sang-Min Choi [1,2,†], Jiho Park [1,3,†], Kiyoung Jang [1] and Chihyun Park [4,*]

1 Department of Computer Science, Yonsei University, 03722 Seoul, Korea; jerassi@yonsei.ac.kr (S.-M.C.); jiho.park@lgcns.com (J.P.); wall2010@yonsei.ac.kr (K.J.)
2 Research Team, The Level Inc., 06141 Seoul, Korea
3 CTO, LG CNS, 07796 Seoul, Korea
4 Department of Computer Science and Engineering, Kangwon National University, 24341 Gangwon-do, Korea
* Correspondence: chihyun@kangwon.ac.kr
† These authors contributed equally to this work.

Received: 20 August 2020; Accepted: 21 September 2020; Published: 1 October 2020

Abstract: A distributed system guarantees the acceptance of Byzantine fault tolerance (BFT) for information transmission. Practical BFT (pBFT) and asynchronous BFT (aBFT) are among the various available forms of BFT. Distributed systems generally share information with all participating nodes. After information is shared, the systems reshare it. Thus, ensuring BFT consumes a considerable amount of time. Herein, we propose Decision search protocols that apply the gossip protocol, denoted by DecisionBFT, for distributed networks with guaranteed BFT. Each protocol in DecisionBFT is completely asynchronous and leaderless; it has an eventual consensus but no round-robin or proof-of-work. The core concept of the proposed technology is the consensus structure, which is based on the directed acyclic graph (DAG) and gossip protocol. In the most general form, each node in the algorithm has a set of k neighbors of most preference. When receiving transactions, a node creates and connects an event block with all its neighbors. Each event block is signed by the hashes of the creating node and its k peers. The consensus structure of the event blocks utilizes a DAG, which guarantees aBFT. The proposed framework uses Lamport timestamps and concurrent common knowledge. Further, an example of a Decision search algorithm, based on the gossip protocol DecisionBFT, is presented. The proposed DecisionBFT protocol can reach a consensus when 2/3 of all participants agree to an event block without any additional communication overhead. The DecisionBFT protocol relies on a cost function to identify the k peers and generate the DAG-based consensus structure. By creating a dynamic flag table that stores connection information between blocks, the gossip protocol achieves a consensus in fewer steps than that in the case of the existing aBFT protocol.

Keywords: consensus algorithm; byzantine fault tolerance; gossip protocol; continuous common knowledge; lamport timestamp; directed acyclic graph

1. Introduction

Beyond the success of cryptocurrencies, blockchain has recently emerged as a technology platform that offers secure, decentralized, and consistent transaction ledgers and has driven innovation across domains including in financial systems, supply chains, and health care. Despite the high demand for distributed ledger technology [1], commercialization opportunities have been obstructed by long processing times for consensus and high power consumption. These issues have been addressed in consensus algorithms such as [2–5].

Byzantine fault tolerance (BFT) that ensures information sharing of two-thirds of participation nodes is usually addressed in distributed database systems [6]. In distributed database systems, we can consider that the proof of Byzantine fault tolerance is equivalent to the purpose of the consensus algorithm. It means that the consensus algorithm guarantees the integrity of transactions between participation nodes in distributed systems [7,8]. Byzantine consensus is not guaranteed for deterministic, completely asynchronous systems with unbounded delays [9]. However, achieving consensus is feasible for nondeterministic systems with a probability of one.

Several approaches have been proposed to reach consensus in distributed systems. The original Nakamoto consensus protocol in Bitcoin uses Proof-of-Work (PoW), in which it is computationally intensive to generate the blocks by participants [10]. Alternative schemes, such as the Proof-of-Stake (PoS) [11], use participants' stakes to generate the blocks. Another approach utilizes directed acyclic graphs (DAG) [4,5,12–14] to facilitate consensus.

Tangle [15], Byteball [16], and Hashgraph [17] are examples of DAG-based consensus algorithms. Tangle selects the blocks to connect in the network, utilizing the accumulated weight of the nonce and Monte Carlo Markov Chain (MCMC). Byteball generates a main chain from the DAG and reaches consensus by using the index information of the chain. Hashgraph connects each block from a node to another random node and searches for whether 2/3 of the members can reach each block and provides proof of BFT via graph search.

1.1. Motivation

Practical Byzantine Fault Tolerance (pBFT) allows all nodes to successfully reach an agreement for a block (information) when a Byzantine node exists [18]. In pBFT, a consensus is reached after all participants share a created block and reshare the share information [19,20]. After consensus is achieved, the block is added to the participants' chains [18,21]. Currently, the time complexity of pBFT is considered to be $O(N^4)$.

HashGraph [17] proposes "gossip about gossip" and virtual voting to reach consensus. HashGraph has several limitations. First, the algorithm operates on a known network, which requires full awareness of all authoritative participants. Second, gossip propagation is slow, and latency increases to $O(n)$ with n participants. Third, it remains unclear whether virtual voting is faster than the chain weight, also known as the longest chain/proof of work concept. These issues are gossip problems rather than consensus problems.

We are interested in a new approach to address the aforementioned issues in methods based on pBFT [18–20] and HashGraph [17]. Specifically, we propose a new consensus algorithm that addresses the following questions: (1) can we reach local consensus in a k-cluster faster for certain values of k? (2) Can we accelerate the spreading of gossip, such as using a broadband-based gossip subset? (3) Can continuous common knowledge be used for consensus decisions with high probability? (4) Can complex decisions be reduced to a binary value consensus?

In this paper, we propose a new approach that can quickly search for Byzantine nodes within the block DAG. In particular, we introduce a new consensus protocol we named the Decision search protocol. The core idea of this protocol is to use a new DAG-based structure, the consensus structure, which allows a faster path search for consensus. We then propose an example of the protocol class, which is referred to as the Decision search protocol, DecisionBFT.

1.2. Generic Framework of DecisionBFT Protocols

We introduce a generic framework of Decision search protocols, DecisionBFT. The Decision search protocol is a DAG-based asynchronous non-deterministic protocol that guarantees aBFT. We propose a consensus structure for a new DAG structure for faster consensus. The consensus protocol generates each block asynchronously, and the consensus algorithm achieves consensus by confirming the number of nodes that know the blocks using the structure. Figure 1 shows an example of the consensus structure that was constructed by using the Decision search protocol. This structure is the result of guaranteed finality. In other words, the color is grouped among events that can verify more than 2/3 of different stages after completing the entire process until re-selection.

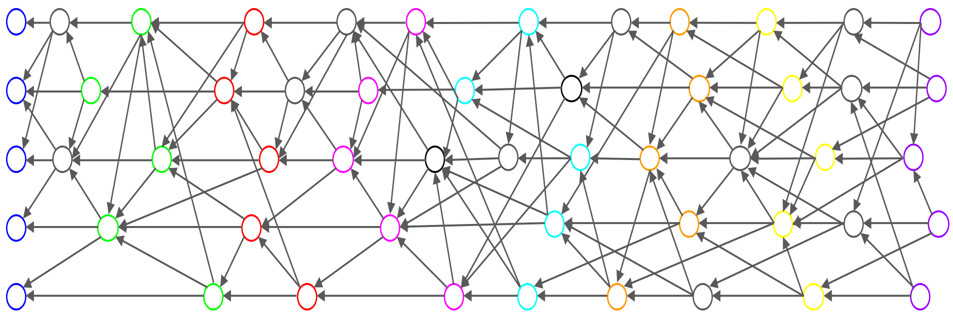


Figure 1. Example of a consensus structure.

The Decision search protocol is based on the following main concepts:

- Event block: All nodes can create event blocks at time t. The structure of an event block includes the signature, generation time, transaction history, and hash information to references. The information of the referenced event blocks can be copied by each node. The first event block of each node is referred to as a leaf event.
- Decision search protocol: Decision search protocol is the set of rules that govern the communication between nodes. When each node creates event blocks, it determines which nodes choose other nodes to broadcast to. Node selection can either be random or occur via a cost function.
- Happened-before: Happened-before is the relationship between nodes that have event blocks. If a path exists from an event block x to y, then x is Happened-before y, which means that the node creating y knows event block x.
- Root: An event block is termed a Root if either (1) it is the first generated event block of a node, or (2) it can reach more than two-thirds of other Roots. Every Root can be a candidate for subDecision.
- Root set: Root set (R_s) is the set of all Roots in the frame. The cardinality of the set is $2n/3 < R_s \leq n$, where n is the number of all nodes.
- Frame: Frame f is a natural number that separates Root sets. The frame increases by 1 in the case of a Root in the new set ($f + 1$). Moreover, all event blocks between the new set and the previous Root set are included in the frame f.
- Flag table: The Flag table stores the reachability from an event block to another Root. The sum of all reachabilities, namely all values in the flag table, indicates the number of reactions from an event block to other Roots.
- Lamport timestamps: For topological ordering, the algorithm responsible for Lamport timestamps uses the happened-before relation to determine the partial order of the entire event block based on logical clocks.
- subDecision: A subDecision is a Root that satisfies being known by more than 2n/3 nodes and more than 2n/3 nodes know the information that is known in nodes. A subDecision can be a candidate for a Decision.
- Decision: A Decision is assigned consensus time by using the Decision search algorithm and is utilized for determining the order between event blocks. Decision blocks allow time consensus ordering and responses to attacks.
- Re-selection: To solve the Byzantine agreement problem, each node reselects a consensus time for a sub-selection, based on the collected consensus time in the Root set of the previous frame. When the consensus time reaches the Byzantine agreement, a subDecision is confirmed as a Decision and is then used for time consensus ordering.

- Consensus structure: The consensus structure is the local view of the DAG held by each node, this local view is used to identify topological ordering, select subDecision, and create time consensus through Decision selection.

As a motivating example, Figure 2 illustrates how consensus is reached via a path search in the consensus structure. In the figure, the leaf set, denoted by R_{s0}, consists of the first event blocks created by individual participant nodes. V is the set of event blocks that belong to neither R_{s0} nor any Root set R_{si}. Given a vertex v in $V \cup R_{si}$, there exists a path from v that can reach a leaf vertex u in R_{s0}. Let r_1 and r_2 be the Root event blocks in Root set R_{s1} and R_{s2}, respectively. Here, r_1 is the block in which a quorum or more blocks exist on a path that reaches a leaf event block. Every path from r_1 to a leaf vertex contains a vertex in V_1. Thus, if there exists a vertex r in V_1 such that r is created by more than a quorum of participants, then r is already included in R_{s1}. Likewise, r_2 is a block that can be reached for R_{s1} including r_1 by using blocks created by a quorum of participants. All leaf event blocks that could be reached by r_1 are connected with more quorum participants through the presence of r_1. The existence of the Root r_2 shows that the information of r_1 is connected with more than a quorum. This kind of path search allows the chain to reach consensus in a similar manner as the aBFT consensus processes. It is essential to keep track of the blocks that satisfy the aBFT consensus process to accelerate path search.

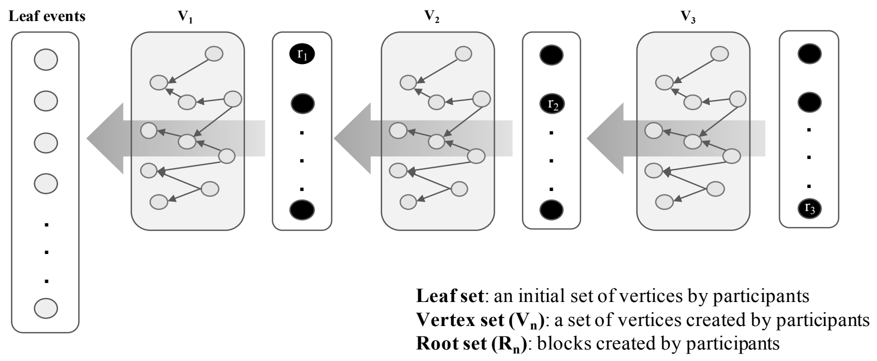

Leaf events
V1
V2
V3
r1
r2
r3
Leaf set: an initial set of vertices by participants
Vertex set (Vn): a set of vertices created by participants
Root set (Rn): blocks created by participants

Figure 2. Consensus method that employs path-search in a directed acyclic graph (DAG) (combines chain with consensus process of asynchronous Byzantine fault tolerance (aBFT)).

1.3. Decision Search Protocol

We now introduce a new specific consensus protocol, namely DecisionBFT. This protocol DecisionBFT is a DAG-based asynchronous non-deterministic protocol that guarantees aBFT. DecisionBFT generates each block asynchronously and uses the consensus structure for faster consensus by verifying the number of nodes that know the blocks.

In this DecisionBFT protocol, we propose several algorithms. In particular, we introduce an algorithm in which a node can identify lazy participants from among its cost-effective peers—say, its k peers. We must stress that a generic Decision search protocol does not depend on any k peer selection algorithm; each node can choose k peers randomly. Each message created by a node is then signed by the creating node and its k peers. We also introduce a flag table data structure that stores the connection information of event blocks. The flag table allows us to quickly traverse the consensus structure to find reachability between event blocks.

The consensus structure can be used to optimize the path search. By using certain event blocks (Root, subDecision, and Decision), a set of Decisions can maintain reliable information between event blocks and reach consensus. Generating event blocks via the Decision search protocol, the consensus structure is updated frequently and can respond strongly to attack situations such as forking and parasite attack. Further, using the flag table over the consensus structure, consensus can be quickly reached, and the ordering between specific event blocks can be determined.

1.4. Contributions

In summary, this study makes the following contributions:

- We propose new consensus protocols, namely Decision search protocols. We introduce the consensus structure for faster consensus.
- We define the topological ordering of nodes and event blocks in the consensus structure. Lamport timestamps are used to introduce more intuitive and reliable ordering in distributed systems. We introduce a flag table at each top event block to improve Root detection.
- We present proof of the way in which a DAG-based protocol can implement concurrent common knowledge for consistent cuts.
- The Decision search protocols allow for faster node synchronization with k-neighbor broadcasts.
- A specific Decision search protocol DecisionBFT is then introduced with specific algorithms. The benefits of the Decision search protocol DecisionBFT include (1) the Root selection algorithm via the flag table, (2) an algorithm to build the DAG, (3) an algorithm for k peer selection via a cost function; (4) faster consensus selection via k peer broadcasts;

The remainder of this paper is organized as follows. In Section 2, we provide an overview of blockchain-related work as well as existing DAG-based protocols. We also describe the preliminaries of our protocols. Section 3 describes our new consensus protocol (Decision search protocol). Section 4 provides proof of Byzantine fault tolerance for our consensus protocol (Decision search protocol). Section 5 presents the consensus algorithm (Decision search algorithm). Several discussions about the Decision search protocols are presented in Section 6. Section 7 concludes the paper with suggestions for future work.

2. Background

2.1. Related Work

2.1.1. Consensus Algorithms

In a consensus algorithm, all participant nodes of a distributed network share transactions and agree with the integrity of the shared transactions [6]. This is equivalent to proving the Byzantine fault tolerance (BFT) in distributed database systems [7,8]. Practical BFT (pBFT) allows all nodes to successfully reach an agreement for a block when a Byzantine node exists [18].

Numerous consensus algorithms have been proposed [2,3]. Proof-of-Work (PoW) requires large amounts of computational work to generate the blocks [10]. Proof-of-Stake (PoS) [11] used participants' stakes and delegated participants' stakes to generate the blocks. Alternative schemes have been proposed to improve algorithms using directed acyclic graphs (DAG) [12]. These DAG-based approaches utilize graph structures to decide consensus; blocks and connections are considered as vertices and edges, respectively.

2.1.2. DAG-Based Approaches

IOTA [15] published a DAG-based technology known as Tangle. The Tips concept was used to address scalability issues within the limitations of the Internet of Things. In addition, a nonce that uses the weight level was composed to achieve the transaction consensus by setting the user's difficulty. To solve the double-spending problem and parasite attack, they used the Markov chain Monte Carlo (MCMC) tip selection algorithm, which randomly selects tips based on the size of the accumulated transaction weights. However, if a transaction conflicts with another, there is still a need to examine all past transaction history to find the conflict.

Byteball [16] uses an internal pay system based on bytes. This is used to pay for adding data to the distributed database. Storage units are all connected to each other, which includes hashes of previous storage units. In particular, the consensus ordering is composed by selecting a single main chain, which is determined as a Root consisting of the most Roots. A majority of Roots detects the double-spend attempts through the consensus time of the main chain. The fee is charged according to the size of the bytes, and requires a list of all units to be searched and updated in the process of determining the Roots.

RaiBlocks [22] have been developed to lower high fees and slow transaction processing. It is a process of obtaining consensus by using a balance-weighted vote on conflicting transactions. Each node participating in the network becomes the principal and manages its data history locally. However, because RaiBlocks generate transactions in a similar way to the anti-spam tool of PoW, all nodes must communicate to create transactions. In terms of scalability, steps would need to be inserted to verify the entire history of transactions when a new node is added.

Hashgraph [17] is an asynchronous DAG-based distributed ledger. Each node is connected by its own ancestor and randomly communicates known events through a gossip protocol. At this time, any famous node can be determined by the see and strong-see relationship at each round to reach a consensus quickly. They state that if more than 2/3 of the nodes reach consensus for an event, it will be assigned a consensus position.

Conflux [14] is a DAG-based Nakamoto consensus protocol. Conflux is a fast, scalable, and decentralized blockchain system that optimistically processes concurrent blocks without discarding any as forks. The Conflux protocol achieves consensus on the total order of the blocks. The total order of the transactions is determined by all participants in the network. Conflux can tolerate up to half of the network as malicious, while the BFT-based approaches can only tolerate up to one-third of malicious nodes.

Parsec [13] proposed a consensus algorithm that guarantees Byzantine faults tolerance in a random synchronous network. Similar to Hashgraph [17], there is no leader. In the algorithm, there is no round-robin and no proof-of-work either. The consensus occurs with a probability of one. Unlike Hashgraph, it provides high-speed processing even in the presence of faults. The parsec algorithm reaches BFT consensus with very weak synchrony assumptions. The algorithm delivers the messages with random delays and the average delay is finite. It allows up to one-third of Byzantine (arbitrary) failures.

Phantom [5] is a protocol based on PoW for a permissionless ledger with a DAG concept. To set higher throughput, PHANTOM can adjust the level of tolerance. For this, the algorithm uses parameter k to control the blocks created concurrently. PHANTOM utilizes a greedy algorithm on the graph structures to identify between blocks by honest nodes and others. This distinction endows PHANTOM with a robust total order of the blocks that is eventually agreed upon by all honest nodes.

Specter [4] is a protocol for cryptocurrencies. It is based on PoW and relies on DAG to generalize blockchain structures. It remains secure from attackers with up to 50% of the computational power even under high throughput and fast confirmation times. The SPECTRE protocol satisfies weaker properties than classic consensus requires. In SPECTRE, the order between any two transactions can be decided from transactions performed by honest users. This is different from the conventional paradigm in which the order must be decided by all non-corrupt nodes.

Blockmania [21], is a mechanism to achieve consensus with several advantages over the more traditional pBFT protocol and its variants. In Blockmania, nodes in a quorum only emit blocks linking to other blocks, irrespective of the consensus state machine. The resulting DAG of blocks (block DAG) is later interpreted to ensure consensus safety, finality, and liveliness. The resulting system has a communication complexity of $O(N^2)$ even in the worst case, and low constant factors, as compared to $O(N^4)$ for pBFT.

Table 1 shows the highlight of related studies for DAG-based approaches

Table 1. Recent studies on DAG-based approaches.

Approach	Description	Limitation
IOTA [15]	The Markov chain Monte Carlo (MCMC) tip selection algorithm and DAG-based techniques	Time consuming to detect conflicts
Byteball [16]	Consensus ordering is composed by selecting a single main chain, which is determined as a Root consisting of the most Roots.	Time consuming to reach consensus
RaiBlocks [22]	A process of obtaining consensus through the balance weighted vote on conflicting transactions.	Time consuming to reach consensus
Hashgraph [17]	Each node is connected by its own ancestor and randomly communicates known events through a gossip protocol.	Each node maintains large information
Conflux [14]	DAG-based Nakamoto consensus protocol.	Time consuming to reach consensus
Parsec [13]	An consensus algorithm that guarantees Byzantine faults tolerance in a random synchronous network.	It allows up to one-third Byzantine (arbitrary) failures.
Phantom [5]	A protocol based on PoW for a permission-less ledger with a DAG concept.	Time consuming to reach consensus
Specter [4]	A protocol based on PoW and DAG to generalize blockchain structures.	Time consuming to reach consensus
Blockmania [21]	A mechanism to achieve consensus with several advantages over the more traditional pBFT protocol and its variants.	Only provides communication efficiency, not consensus protocol.

2.1.3. Lamport Timestamps

Lamport [23] defines the "happened-before" as the relation between any pair of events in a distributed system of machines. The happened-before relation, denoted by $\rightarrow$, is defined without using physical clocks to partially order events in the system. The relation "$\rightarrow$" satisfies the following three conditions: (1) if v and v' are events in the same process, and v precedes v', then $v \rightarrow v'$; (2) if v is the action for sending a message by one process and v' is the receipt of the sent message by another process, then $v \rightarrow v'$; (3) if $v \rightarrow v'$ and $v' \rightarrow v''$ then $v \rightarrow v''$. Two distinct events v and v' are considered to be concurrent if $v \nrightarrow v'$ and $v' \nrightarrow v$. The happens-before relation can be viewed as a causality effect: that $v \rightarrow v'$ implies that event v may causally affect event v'. If one cannot causally affect another, two events are concurrent.

Lamport introduces logical clocks, which are a way of assigning a number to an event. A clock C_i for each process P_i is a function that assigns a number $C_i(v)$ to any event $v \in P_i$. The entire system of blocks is represented by the function C that assigns to any event v the number $C(v)$, where $C(v) = C_j(v)$ if b is an event in process P_j. The Clock Condition states that for any events v, v': if $v \rightarrow v'$, then $C(v)$ $< C(v')$.

To satisfy the Clock Condition, the clocks must satisfy two conditions. First, each process i increments C_i between any two successive events. Second, each message m contains a timestamp T_m, which means the time at which the message is sent. After receiving a message with T_m, a process advance the clock to be later than T_m.

Given any arbitrary total ordering $\prec$ of the processes, the total ordering $\Rightarrow$ is defined as follows: if v is an event in process P_i and v' is an event in process P_j, then $v \Rightarrow v'$ if and only if either (i) $C_i(v) < C_j(v')$ or (ii) $C(v) = Cj(v')$ and $P_i \prec P_j$. The Clock Condition implies that if $v \rightarrow v'$, then $v \Rightarrow v'$.

2.1.4. Concurrent Common Knowledge

Panangaden and Taylor [24] defined a model to explain the concurrent common knowledge (CCK) in asynchronous distributed systems. The system consists of a series of processes that can communicate only by transmitting messages along a fixed set of channels. The network is not necessarily fully connected. The system is asynchronous in that it does not include a global clock, and the relative speed of the process is independent. The delivery time for a message is finite but unbounded.

The local state of a process is denoted by s_i^j. Actions are state transformers; an action is a function from local states to local states. An action can be either a send(m) action, where m is a message, a receive(m) action, and an internal action. h_i is a local history of process i. It is a (possibly infinite) sequence of alternating local states starting with distinct initial states and actions. We write this sequence as follows: $h_i = s_i^0, \alpha_i^1, s_i^1, \alpha_i^2, s_i^2, \alpha_i^3 ...$, where s_i^j is the j-th state (action) in the local history of process i.

An asynchronous system consists of the following sets: a set P = {1,...,N} of process identifiers; a set $C \subseteq \{(i,j)$ s.t. $i, j \in P\}$ of channels. A set H_i of possible local histories for each process; a set A of asynchronous runs, where each run is $\sigma = \langle h_1, h_2, h_3, ...h_N \rangle$, and a set M of messages.

The CCK model of an asynchronous system does not mention time, but rather, events are ordered based on Lamport's happens-before relation. They use Lamport's theory to describe the global states of an asynchronous system. They use the concepts of the global-state and consistent-run to define a model of concurrent common knowledge.

With this paper, we are the first to define semantics for DAG-based protocols. In particular, we use concurrent common knowledge [24] as the basis for explaining the BFT of asynchronous distributed systems.

2.2. Preliminaries

The history of a Decision search protocol can be represented by a DAG, $G = (V, E)$, where V is a set of vertices and E is a set of edges. Each vertex in a row (node) represents an event. Time flows from left-to-right along the graph, thus the left vertices represent events that occurred earlier in history. A path p in G is a sequence of vertices $(v_1, v_2, \ldots, v_k)$ obtained by following the edges in E. Let v_c be a vertex in G. A vertex v_p is the parent of v_c if there is an edge from v_p to v_c. A vertex v_a is an ancestor of v_c if there is a path from v_a to v_c.

Definition 1 (node)**.** *Each machine that participates in the Decision search protocol is referred to as a node.*

Let n denote the total number of nodes. In Decision search protocols, we assume that n is a fixed number.

Definition 2 (event block)**.** *Each node can create event blocks and send (receive) messages to (from) other nodes.*

Definition 3 (vertex)**.** *An event block is a vertex of the consensus structure.*

Suppose a node n_i creates an event v_c after an event v_s in n_i. Each event block has exactly k references, one of which is self-reference, and the other $k - 1$ references point to the top events of the $k - 1$ peer nodes of n_i.

Definition 4 (peer node)**.** *A node n_i has k peer nodes.*

Definition 5 (top event)**.** *An event v is the top event of a node n_i if no other event in n_i references v.*

Definition 6 (self-ref)**.** *An event v_s is termed the "self-ref" of event v_c if the self-ref hash of v_c points to the event v_s. We denote $v_c \hookrightarrow^s v_s$.*

Definition 7 (ref). *An event v_r is termed the "ref" of event v_c if the reference hash of v_c points to the event v_r. We denote $v_c \hookrightarrow^r v_r$.*

For simplicity, we can use $\hookrightarrow$ to denote a reference relationship (either $\hookrightarrow^r$ or $\hookrightarrow^s$).

Definition 8 (self-ancestor). *An event block v_a is the self-ancestor of an event block v_c if there exists a sequence of events such that $v_c \hookrightarrow^s v_1 \hookrightarrow^s \ldots \hookrightarrow^s v_m \hookrightarrow^s v_a$. We denote $v_c \hookrightarrow^{sa} v_a$.*

Definition 9 (ancestor). *An event block v_a is an ancestor of an event block v_c if there exists a sequence of events such that $v_c \hookrightarrow v_1 \hookrightarrow \ldots \hookrightarrow v_m \hookrightarrow v_a$. We denote $v_c \hookrightarrow^a v_a$.*

For simplicity, we simply use $v_c \hookrightarrow^a v_s$ to refer to both ancestor and self-ancestor relationships, unless we need to distinguish between the two cases.

Definition 10 (Consensus structure). *The consensus structure is a DAG graph $G = (V, E)$ consisting of V vertices and E edges. Each vertex $v_i \in V$ is an event block. An edge $(v_i, v_j) \in E$ refers to a hashing reference from v_i to v_j; that is, $v_i \hookrightarrow v_j$.*

Domination Relation

Then, we define the domination relation for the event blocks. To begin with, we first introduce pseudo-vertices, top and bot, of the DAG consensus structure G.

Definition 11 (pseudo top). *A pseudo-vertex, termed the top, is the parent of all top event blocks. Denoted by $\top$.*

Definition 12 (pseudo bottom). *A pseudo-vertex, termed the bottom, is the child of all leaf event blocks. Denoted by $\bot$.*

With the pseudo-vertices, we have that $\bot$ happened before all event blocks. In addition, all event blocks happened before $\top$. That is, for all events v_i, $\bot \to v_i$, and $v_i \to \top$.

Definition 13 (dom). *An event v_d dominates an event v_x if every path from $\top$ to v_x is required to pass through v_d. We denote $v_d \gg v_x$.*

Definition 14 (strict dom). *An event v_d strictly dominates an event v_x if $v_d \gg v_x$ and v_d does not equal v_x. We denote $v_d \gg^s v_x$.*

Definition 15 (domfront). *A vertex v_d is said to be a vertex v_x if v_d dominates an immediate predecessor of v_x, but v_d does not strictly dominate v_x. We denote $v_d \gg^f v_x$.*

Definition 16 (dominance frontier). *The dominance frontier of a vertex v_d is the set of all nodes v_x such that $v_d \gg^f v_x$. Denoted by $DF(v_d)$.*

From the above definitions of the domfront and dominance frontier, the following holds. If $v_d \gg^f v_x$, then $v_x \in DF(v_d)$.

Here, we introduce a new idea that extends the concept of domination.

Definition 17 (subgraph). *For a vertex v in a DAG G, let $G[v] = (V_v, E_v)$ denote an induced-subgraph of G such that V_v consists of all ancestors of v including v, and E_v is the induced edges of V_v in G.*

For a set S of vertices, an event v_d $\frac{2}{3}$-dominates S if there are more than $2/3$ of vertices v_x in S such that v_d dominates v_x. Recall that R_1 is the set of all leaf vertices in G. The $\frac{2}{3}$-dom set D_0 is the same as the set R_1. The $\frac{2}{3}$-dom set D_i is defined as follows:

Definition 18 ($\frac{2}{3}$-dom set)**.** *A vertex v_d belongs to a $\frac{2}{3}$-dom set within graph $G[v_d]$, if v_d $\frac{2}{3}$-dominates R_1. The $\frac{2}{3}$-dom set D_k consists of all Roots d_i such that $d_i \notin D_i$, $\forall$ i = 1..(k-1), and d_i $\frac{2}{3}$-dominates D_{i-1}.*

Lemma 1. *The $\frac{2}{3}$-dom set D_i is the same as the Root set R_i for all nodes.*

Table 2 contains the notations of symbols used in definitions, propositions, and lemmas.

Table 2. Notations of symbols used in definitions, propositions, and lemmas.

Symbol	Name	Definition	Example
$\hookrightarrow^a$	ref	reference hash of an event block points to another event block	$v_c \hookrightarrow^a v_r$
$\top$	pseudo-top	parent of all top event blocks	$\top$
$\bot$	pseudo-bottom	child of all leaf event blocks	$\bot$
$\mapsto$	Happened-Immediate-Before	if v_x is a (self-) ref of v_y	$v_x \mapsto v_y$
$\rightarrow$	Happened-Before	if v_x is a (self-) ancestor of v_y	$v_x \rightarrow v_y$
$\parallel$	concurrent	Two event blocks are said to be concurrent if neither of them happened before the other	$v_x \parallel v_y$
$\pitchfork$	fork	If two event blocks have the same creator	$v_x \pitchfork v_y$

2.3. Examples of Dominant Relation in Consensus Structure

This section provides several examples of consensus structures and the dominant relation between their event blocks.

Figure 3 shows examples of consensus structure and dominator trees. The dominator tree is based on *Dominator* in graph theory. For example, any node d dominating a node n should refer to d. In terms of the notation, this is written as d dom n. (a) and (b) in Figure 3 are representations of their connection as events occur over time. Based on the Dominator, event node $A2$ acts as a dominator from other nodes. However, Figure 4 explains the case where only 2n/3 or more of the requirements of the dom set are satisfied. Figure 4 depicts an example of a consensus structure and 2/3 dom sets. Suppose the connection between event nodes with direct graphs can exchange information with each other. $A1$ contains four of the five dominant groups $a0, b0, c0, e0$ and $C1$ contains groups $a0, b0, c0, e0$, and so on. Therefore, if the top event node is connected to $A1$ and $C1$, the top can be considered to dominate more than 2/3 of the total dom sets.

Figure 5 shows an example of dependency graphs. In each row, the left-most figure shows the latest consensus structure. The figures on the left in each row depict the dependency graphs of each node, which are in their compact form. In the absence of a fork, each of the compact dependency graphs is a tree. The nodes from A to E are represented in their respective views, and the graph that combines all the nodes is shown on the far left. Panels (a)–(c) are representations of their connection as events occur over time. Figure 5 shows the updated graph that reflects the events of nodes A and E. Because each node has a graph from its own perspective, each node has its own differences.

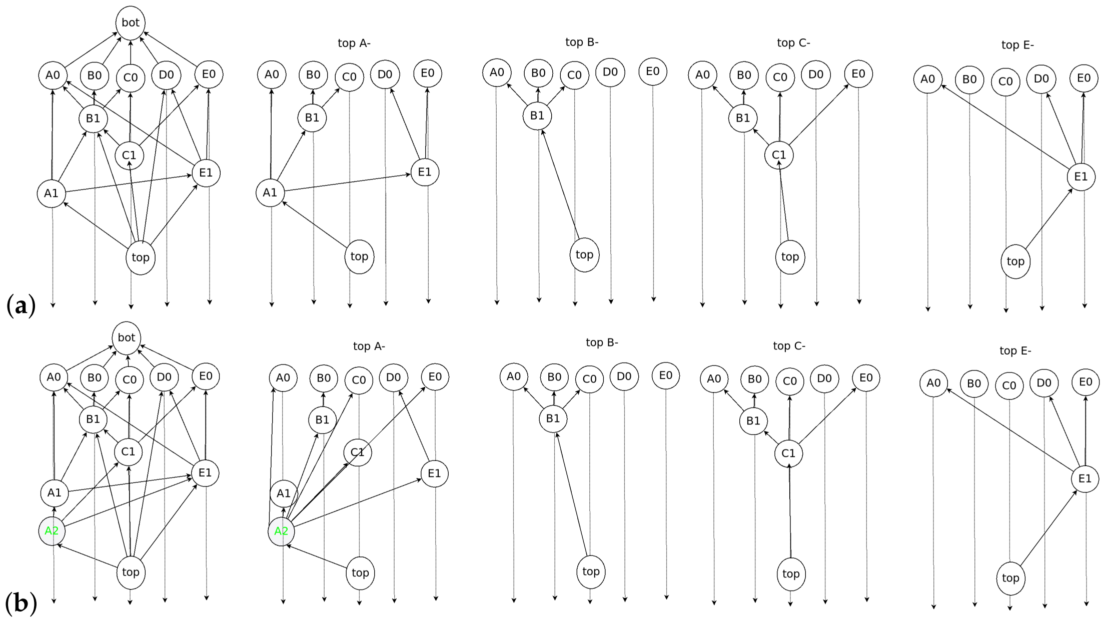


Figure 3. Examples of consensus structure and dominator tree. (**a**,**b**) are representations of their connection as events occur over time.

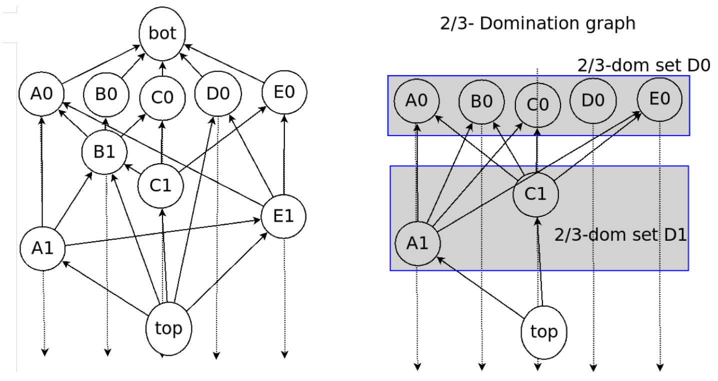


Figure 4. Example of consensus structure and its 2/3 domination graph. The $\frac{2}{3}$-dominant sets are shown in gray.

Figure 6 shows an example of a pair of fork events. Each row shows a consensus structure (left most) and the compact dependency graphs on each node (right). The red and green vertices represent fork events. For example, suppose that node y and y′ participate in red ($A1$) and green events ($A2$), they do not refer to each other. Thus, they judge the event to differ from node A without the consensus of more than 2n/3 of all the nodes. When a fork occurs, it can be regarded as an activity of a malicious node and can be identified by the consensus structure.

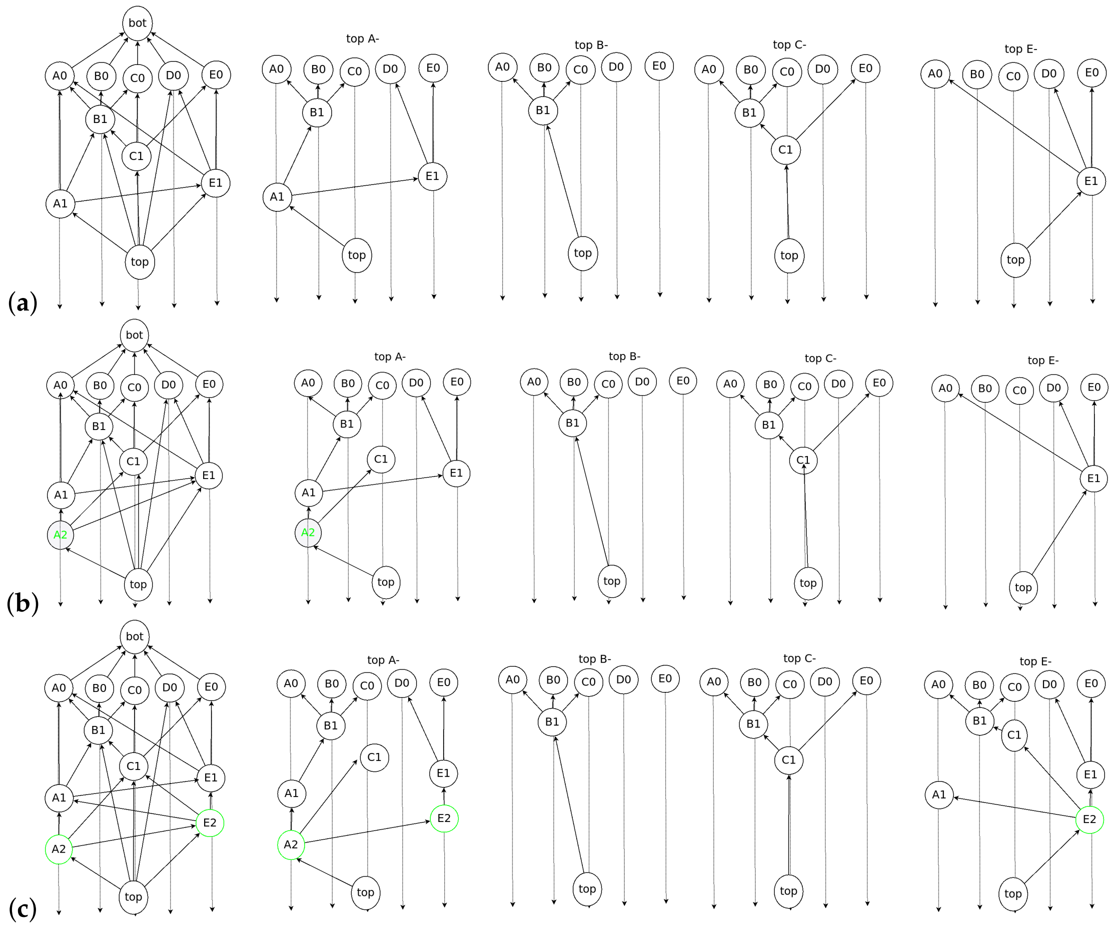


Figure 5. Example of dependency graphs on individual nodes. From (**a**)–(**c**), one new event block is appended. The graphs do not fork, and the simplified dependency graphs become trees.

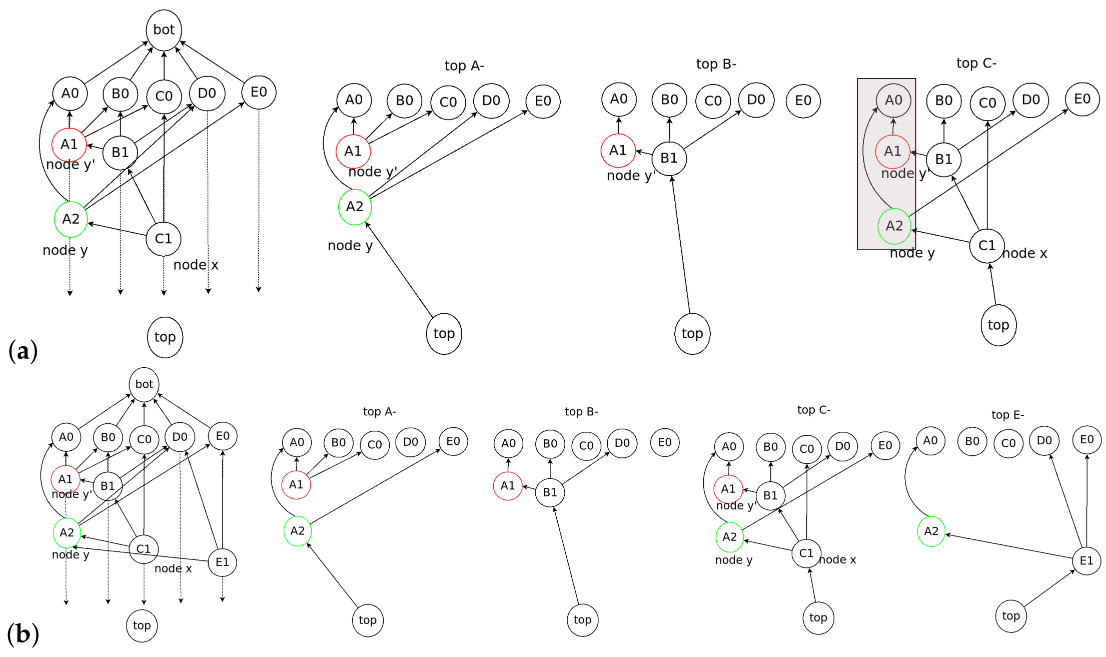
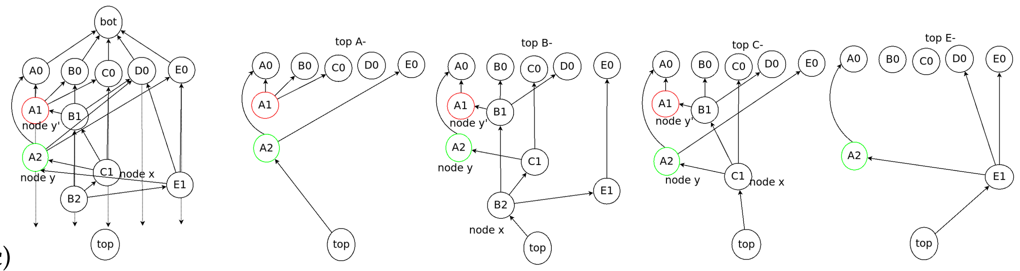


Figure 6. *Cont.*



(**c**)

Figure 6. Example of a pair of fork events in a consensus structure. The fork events are shown in red and green. The consensus structures from (**a**)–(**c**) differ by adding one single event at a time.

3. Generic Framework of Decision Search Protocol

This section describes the key concepts of our new family of consensus protocols.

3.1. Consensus Structure

The core idea of a decision search protocol is to explore a DAG-based structure, known as a consensus structure. In this protocol, a (participant) node is a server (machine) of the distributed system. Each node can create messages, send messages to, and receive messages from other nodes. Communication between nodes is asynchronous.

The consensus structure consists of event blocks, including user information and edges between event blocks. The event blocks are created by a node after the node communicates information to another node. The consensus structure comprises event blocks as vertices and block communication as edges.

Let n be the number of participant nodes. For consensus, the algorithm examines whether an event block is shared with 2n/3 nodes, where n is the number of all nodes. The happen-before of event blocks with 2n/3 nodes means that more than two-thirds of all nodes in the consensus structure knows the event block.

3.2. Decision Search Algorithm

A decision search algorithm (DSA) is presented. The DSA, a consensus algorithm for solving the byzantine agreement problem, uses Root, subDecision, and Decision blocks to find the consensus time for event blocks. Algorithm 1 shows the pseudocode of a consensus structure. The algorithm consists of two parts, which are executed in parallel.

In the first part, each node requests synchronization and creates an event block. In line 3, a node runs the node selection algorithm. The node selection algorithm returns the k IDs of other nodes to communicate with. In lines 4 and 5, the node synchronizes with other nodes. Line 6 runs the event block creation, at which step the node creates an event block and checks whether it is Root. Then, the node broadcasts the created event block to other nodes in line 7. The step in this line is optional. In lines 8 and 9, subDecision selection and Decision time consensus algorithms are invoked. The algorithms determine whether the specified Root can be a subDecision, assign the consensus time, and then confirm the Decision.

The second part is designed to respond to synchronization requests. In lines 10 and 11, the node receives a synchronization request and then sends its response to the consensus structure.

Algorithm 1 shows the main procedure of our proposed framework. In this procedure, we first execute the k-node selection algorithm. Algorithm 2 shows the details of the process for k-node selection. After the k-selection algorithm, the nodes in the framework communicate synchronized information with each other. Then, we create a new event block and proceed with the three-step process for consensus: Root selection, subDecision selection, and Decision time consensus. We repeat this process, generate new event blocks, and reach a consensus.

Algorithm 1 Main Procedure

```
1: procedure MAIN PROCEDURE
2: loop:
3:     A, B = k-node Selection algorithm()
4:
5:     Request sync to node A and B
6:     Sync all known events by Decision search protocol
7:     Event block creation
8:     (optional) Broadcast the message
9:     Root selection
10:    subDecision selection
11:    Decision time consensus
12: loop:
13:    Request sync from a node
14:    Sync all known events by Decision search protocol
```

Algorithm 2 *k*-neighbor Node Selection

```
1: procedure k-NODE SELECTION
2:     Input: Height Vector $H$, In-degree Vector $I$
3:     Output: reference node $ref$
4:     min_cost ← $INF$
5:     $s_{ref}$ ← None
6:     for $k \in Node\_Set$ do
7:         $c_f \leftarrow \frac{I_k}{H_k}$
8:         if min_cost $> c_f$ then
9:             min_cost ← $c_f$
10:            $s_{ref}$ ← k
11:        else if min_cost equal $c_f$ then
12:            $s_{ref} \leftarrow s_{ref} \cup k$
13:    $ref$ ← random select in $s_{ref}$
```

3.3. Node Structure

This section provides an overview of the node structure in the Decision search protocol.

Each node has a height vector, an in-degree vector, flag table, frames, subDecision checklist, max–min value, and its own consensus structure. In the node structure, height is the number of event blocks created by the *i*-th node, and in-degree denotes the number of edges from other event blocks created by other nodes to the top event block. The top event block indicates the most recently created event block by a node. The flag table is an $n \times k$ matrix, where n is the number of nodes and k is the number of Roots by which an event block can be reached. If an event block e created by the *i*-th node can reach *j*-th Root, then the flag table stores the hash value of *j*-th Root. Each node maintains the flag table of each top event block.

Frames are used to store the Root set in each frame. The subDecision checklist has two types of checkpoints: subDecision candidate (SC) and subDecision (S). If a Root in a frame is a SC, a node checks C, and if a Root becomes a subDecision, a node checks the S part. The max–min value is the timestamp that addresses the Decision selection.

Figure 7 shows an example of the node structure component of a node A. In the figure, each value, excluding the self-height in the height vector, is 1 because the initial state is shared with all the nodes. In the in-degree vector, node A stores the number of edges from other event blocks created by other nodes to the top event block. Therefore, the in-degrees of nodes A, B, and C are 1. In the flag table, node A knows two other Root hashes because the top event block can reach those two Roots. Node A also knows that other nodes know their own Roots. In the example situation, a subDecision candidate and subDecisions do not exist; thus, the subDecision checklist is empty. The max–min value is empty for the same reason as the subDecision checklist. The Root, subDecision, and Decision selection algorithm are introduced in Section 5.

3.4. Event Block Creation

In the Decision search protocol, every node can create an event block. Each event block refers to k other event blocks using their hash values. In a Lachesis protocol, a new event block refers to k-neighbor event blocks under the following conditions:

1. Each of the k reference event blocks is the top event block of its own node.
2. One reference should be made to a self-ref that refers to an event block of the same node.
3. The other $k - 1$ reference refers to the other $k - 1$ top event nodes on other nodes.

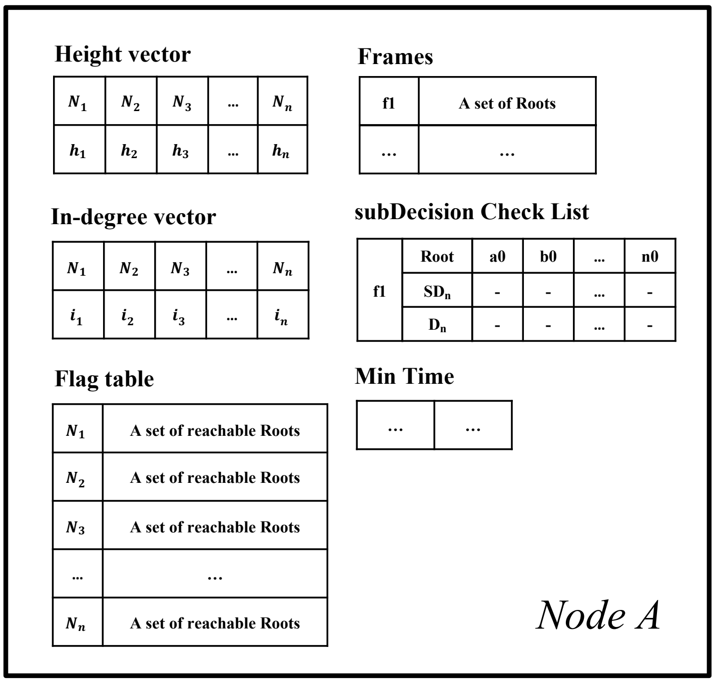
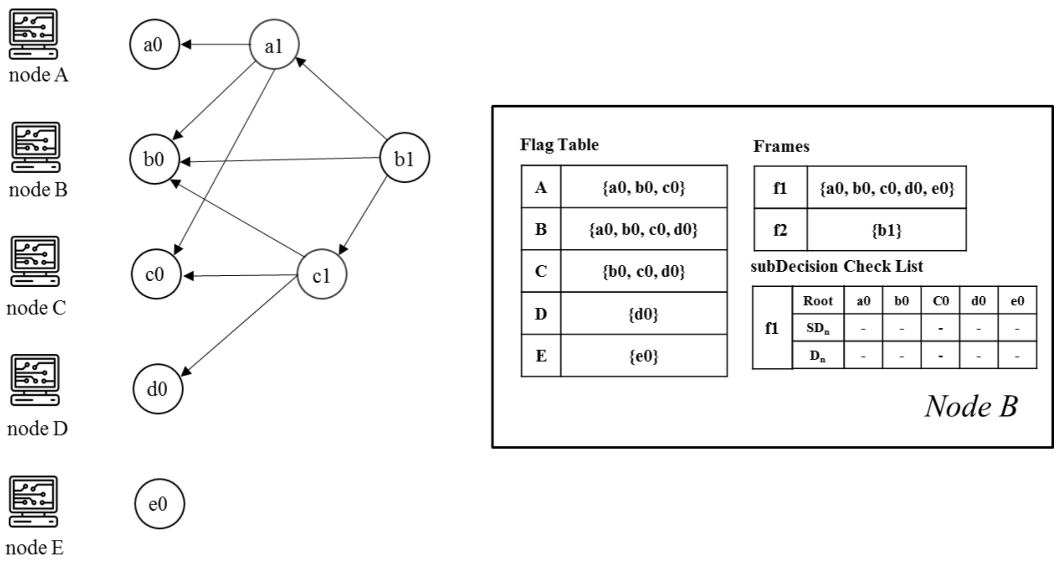

Height vector
N_1 N_2 N_3 ... N_n
h_1 h_2 h_3 ... h_n
Frames
f1 A set of Roots
... ...
In-degree vector
N_1 N_2 N_3 ... N_n
i_1 i_2 i_3 ... i_n
subDecision Check List
f1 Root a0 b0 ... n0
SD_n - - ... -
D_n - - ... -
Flag table
N_1 A set of reachable Roots
N_2 A set of reachable Roots
N_3 A set of reachable Roots
... ...
N_n A set of reachable Roots
Min Time
... ...
Node A

Figure 7. Example of node structure.

The flag table information is updated during the block creation process because the top event block is changed. Figure 8 shows an example of event block creation with a flag table. In this example, the recently created event block is b_1 by node B. The figure shows the node structure of node B. We omit other information, such as height and in-degree vectors because we only focus on the change in the flag table with event block creation in this example. The flag table of b_1 in Figure 8 is updated with the information of the previous connected event blocks a_1, b_0, and c_1. Thus, the set of flag tables is the results of the OR operation among the three Root sets for $a1$ (a_0, b_0, and c_0), b_0 (b_0), and c_1 (b_0, c_0, and d_0).



Flag Table

A	{a0, b0, c0}
B	{a0, b0, c0, d0}
C	{b0, c0, d0}
D	{d0}
E	{e0}

Frames

f1	{a0, b0, c0, d0, e0}
f2	{b1}

subDecision Check List

f1	**Root**	**a0**	**b0**	**C0**	**d0**	**e0**
	SD$_n$	-	-	-	-	-
	D$_n$	-	-	-	-	-

Figure 8. Example of event block creation with the flag table.

3.5. Topological Ordering of Events Using Lamport Timestamps

Every node has a physical clock and requires physical time to create an even block. However, Decision search protocols rely on the logical clock of each node. For this purpose, we use "Lamport timestamps" [23] to determine the time ordering between event blocks in an asynchronous distributed system.

The Lamport timestamps algorithm is as follows:

1. Each node increments its count value before creating an event block.
2. When sending a message that includes its count value, the receiver should consider from which sender the message is received and increment its count value.
3. If the current counter is less than or equal to the received count value from another node, then the count value of the recipient is updated.
4. If the current counter is greater than the count value received from another node, then the current count value is updated.

We use Lamport's algorithm to enforce a topological ordering of event blocks and use it in the Decision selection algorithm. Figure 9 shows an example of Lamport timestamps.

Because an event block is created on the basis of logical time, the sequence between each event block is immediately determined. Because the Lamport timestamps algorithm provides a partial order of all events, the entire time ordering process can be used for BFT.

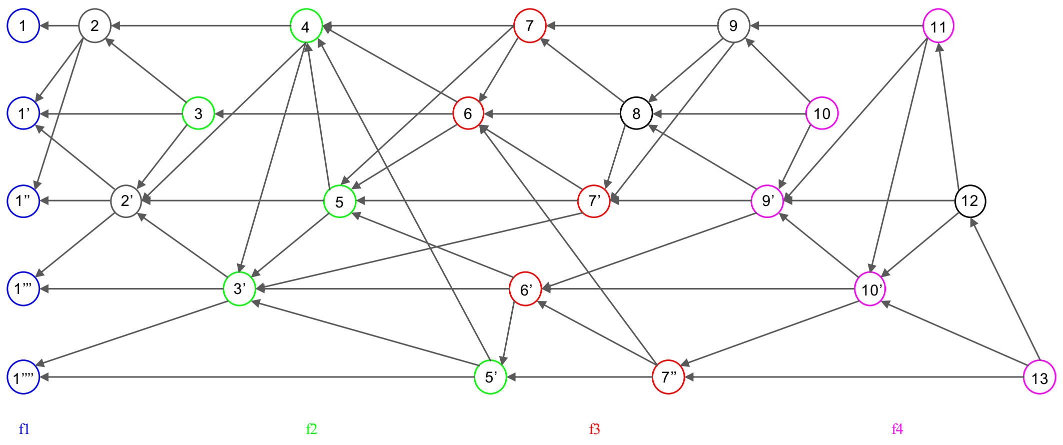


Figure 9. An example of Lamport timestamps.

3.6. Topological Consensus Ordering

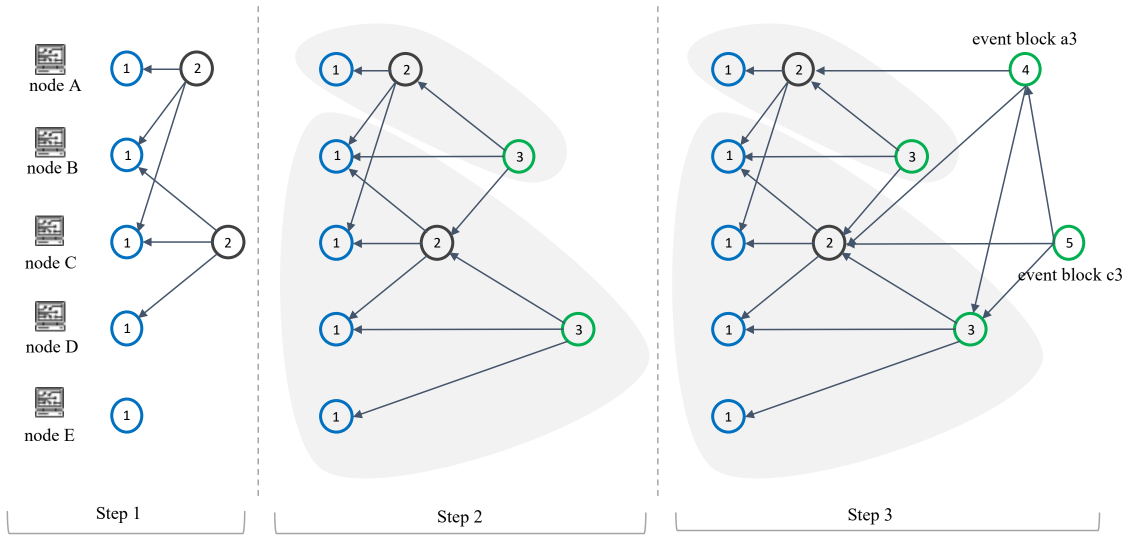
The sequential order of each event block is an important aspect of BFT. To determine the pre-and-post sequence between all event blocks, we use the Decision consensus time, Lamport timestamp algorithm, and hash value of the event block.

First, when each node creates event blocks, they have a logical timestamp based on the Lamport timestamp. This means that they have partial ordering between the relevant event blocks. Each subDecision has a consensus time of Decision. This consensus time is computed based on the logical time nominated by the other nodes at the time of the 2n/3 agreement.

In the DSA, each event block is based on the following three rules to reach an agreement:

1. If there is more than one Decision with different times on the same frame, the event block with smaller consensus time has higher priority.
2. If there is more than one Decision with the same consensus time on the same frame, determine the order based on the own logical time from the Lamport timestamp.
3. When more than one Decision has the same consensus time, if the local logical time is the same, the hash function prioritizes a smaller hash value.

Figure 10 depicts an example of topological consensus ordering. All nodes are represented in a table with each step of the Lamport timestamp. Here, we explain the time ordering method by example. Based on the *a*3 and *c*3 event blocks, we determine the prioritized ordering of event blocks. For example, nodes are required to generate each event block. However, when event blocks refer to different event blocks, they can exchange information with each other. In step 1, all nodes generated the first event block, and they all had a Lamport time of 1. Further, *a* and *c* each generate a second event block, resulting in a Lamport time of 2. Similarly, in step 2, nodes *b* and *d* created a second block of events, but because they refer to *a*2 and *c*2, they have a Lamport time of 3. Likewise, in step 3, nodes *a* and *c* each created a third event block, but each had a different Lamport time. This is because the previous reference values are different. In this way, we established time ordering rules and propose topological ordering.



Node	Lamport Timestamp	
A	a1	1
B	b1	1
C	c1	1
D	d1	1
E	e1	1

Node	Lamport Timestamp			
A	a1	1	a2	2
B	b1	1	b2	3
C	c1	1	c2	2
D	d1	1	d2	3
E	e1	1	e2	-

Node	Lamport Timestamp					
A	a1	1	a2	2	a3	4
B	b1	1	b2	3	b3	-
C	c1	1	c2	2	c3	5
D	d1	1	d2	3	d3	-
E	e1	1	e2	-	e3	-

Figure 10. Example of topological consensus ordering.

Figure 11 shows the part of the consensus structure in which the final consensus order is determined on the basis of these three rules. The number represented by each event block is the logical time based on the Lamport timestamp.

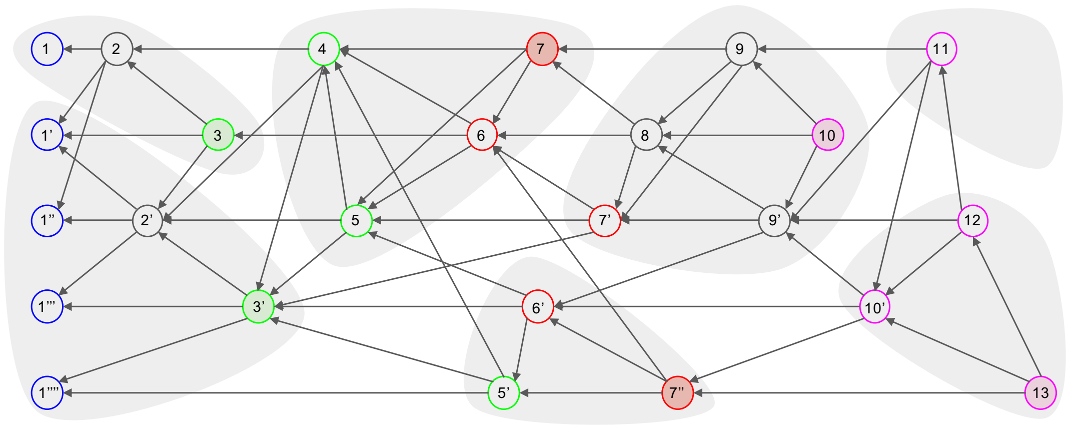


Figure 11. Example of time ordering of event blocks.

The final topological consensus order containing the event blocks is based on the agreement for the Decision. Based on each Decision, the blocks have different colors depending on their range.

3.7. Peer Selection Algorithm

To create an event block, a node needs to select k other nodes. Decision search protocols do not depend on the way in which peer nodes are selected. A simple approach is to use a random selection from the pool of n nodes. Another approach is to define criteria or cost functions to select other peers of a node.

Within a distributed system, a node can select other nodes with low communication costs, low network latency, high bandwidth, and highly successful transactions.

4. Proof of Decision Search Protocol

This section presents proof of the liveness and safety of our Decision search protocols. We aim to show that our consensus is BFT with the presumption that more than two-thirds of participants are reliable nodes. We first provide some definitions, lemmas, and theorems. Then, we validate the BFT.

4.1. Proof of Byzantine Fault Tolerance for Decision Search Algorithm

Definition 19 (Happened-Immediate-Before). *An event block v_x is said to be a Happened-Immediate-Before an event block v_y if v_x is a (self-)ref of v_y. We denote $v_x \mapsto v_y$.*

Definition 20 (Happened-Before). *An event block v_x is said to have Happened-Before an event block v_y if v_x is a (self-)ancestor of v_y. We denote $v_x \to v_y$.*

The happens-before relation is the transitive closure of happens-immediately-before. Thus, an event v_x happened before an event v_y if one of the following occurs: (a) $v_y \hookrightarrow^s v_x$, (b) $v_y \hookrightarrow^r v_x$, or (c) $v_y \hookrightarrow^a v_x$. We propose the following proposition:

Proposition 1 (Happened-Immediate-Before consensus structure). *$v_x \mapsto v_y$ iff $v_y \hookrightarrow v_x$ iff edge $(v_y, v_x) \in E$ of the consensus structure.*

Lemma 2 (Happened-Before Lemma). *$v_x \to v_y$, if $v_y \hookrightarrow^a v_x$.*

Definition 21 (concurrent)**.** *Two event blocks v_x and v_y are said to be concurrent if neither of them happened before the other. We denote $v_x \parallel v_y$.*

Given two vertices v_x and v_y both contained in two consensus structures, G_1 and G_2, on two nodes. We have the following: (1) $v_x \rightarrow v_y$ in G_1 iff $v_x \rightarrow v_y$ in G_2; (2) $v_x \parallel v_y$ in G_1 iff $v_x \parallel v_y$ in G_2.

Below are a few of the main definitions in the Decision search protocol.

Definition 22 (Leaf)**.** *The first created event block of a node is known as a leaf event block.*

Definition 23 (Root)**.** *The leaf event block of a node is Root. When an event block v can reach more than 2n/3 of the Roots in the previous frames, v becomes a Root.*

Definition 24 (Root set)**.** *The set of all first event blocks (leaf events) of all nodes form the first Root set R_1 ($|R_1|$ = n). The Root set R_k consists of all Roots r_i such that $r_i \notin R_i$, $\forall$ i = 1...(k-1) and r_i can reach more than 2n/3 other Roots in the current frame, i = 1...(k-1).*

Definition 25 (Frame)**.** *Frame f_i is a natural number that separates the Root sets.*

The Root set at frame f_i is denoted by R_i.

Definition 26 (consistent structures)**.** *consensus structures G_1 and G_2 are consistent iff for any event v contained in both structures, $G_1[v] = G_2[v]$. We denote $G_1 \sim G_2$.*

When two consistent structures contain the same event v, both structures contain the same set of ancestors for v, with the same reference and self-ref edges between those ancestors:

Theorem 1. *All nodes have consistent consensus structures.*

Proof. If two nodes have consensus structures containing event v, then they have the same k hashes contained within v. A node will not accept an event during a sync unless that node already has k references for that event, thus both consensus structures must contain k references for v. The cryptographic hashes are assumed to be secure; therefore, the references must be the same. By induction, all ancestors of v must be the same. Therefore, the two consensus structures are consistent. □

Definition 27 (creator)**.** *If a node n_x creates an event block v, then the creator of v, denoted by $cr(v)$, is n_x.*

Definition 28 (fork)**.** *The pair of events (v_x, v_y) is a fork if v_x and v_y have the same creator, but neither is a self-ancestor of the other. We denote $v_x \pitchfork v_y$.*

For example, let v_z be an event in node n_1 and two child events v_x and v_y of v_z. If $v_x \hookrightarrow^s v_z$, $v_y \hookrightarrow^s v_z$, $v_x \not\hookrightarrow^s v_y$, $v_y \not\hookrightarrow^s v_z$, then ($v_x$, v_y) is a fork. The fork relation is symmetric; that is, $v_x \pitchfork v_y$ iff $v_y \pitchfork v_x$.

Lemma 3. *$v_x \pitchfork v_y$ iff $cr(v_x) = cr(v_y)$ and $v_x \parallel v_y$, respectively.*

Proof. By definition, (v_x, v_y) is a fork if $cr(v_x) = cr(v_y)$, $v_x \not\hookrightarrow^a v_y$, and $v_y \not\hookrightarrow^a v_x$. Using Happened-Before, the second part means $v_x \not\rightarrow v_y$ and $v_y \not\rightarrow v_x$. By the definition of concurrency, we obtain $v_x \parallel v_y$. □

Lemma 4. *(fork detection). If there is a fork $v_x \pitchfork v_y$, then v_x and v_y cannot both be Roots on honest nodes.*

Proof. Here, we prove by contradiction. Any honest node cannot accept a fork, thus v_x and v_y cannot be Roots on the same honest node. Now, we prove a more general case. Suppose that both v_x is a Root of n_x and v_y is the Root of n_y, where n_x and n_y are honest nodes. Because v_x is a Root, it reached events created by more than 2/3 of the member nodes. Similarly, v_y is a Root, and it reaches events created by more than 2/3 of the member nodes. Thus, there must be an overlap of more than $n/3$ members of those events in both sets. Because we assume that fewer than $n/3$ members are not honest, there must be at least one honest member in the overlap set. Let n_m be such an honest member. Because n_m is honest, n_m does not allow a fork. This contradicts the assumption. Thus, the lemma is proved. □

Each node n_i has a consensus structure G_i. We define a consistent structure from a sequence of consensus structures G_i.

Definition 29 (consistent structure)**.** *A global consistent structure G^C is a structure if $G^C \sim G_i$ for all G_i.*

We denote $G \subseteq G'$ to denote that G is a subgraph of G'.

Lemma 5. $\forall G_i$ *(*$G^C \subseteq G_i$*),*

Lemma 6. $\forall v \in G^C\ \forall G_i$ *(*$G^C[v] \subseteq G_i[v]$*),*

Lemma 7. *(*$\forall v_c \in G^C$*) (*$\forall v_p \in G_i$*) ((*$v_p \rightarrow v_c$*)* $\Rightarrow v_p \in G^C$*).*

Now, we state the following important propositions.

Definition 30 (consistent Root)**.** *Two structures G_1 and G_2 are Root consistent; if for every v contained in both structures, and v is a Root of the j-th frame in G_1, then v is a Root of the j-th frame in G_2.*

Proposition 2. *If $G_1 \sim G_2$, then G_1 and G_2 are Root-consistent.*

Proof. By consistent structures, if $G_1 \sim G_2$ and v belong to both structures, then $G_1[v]$ = $G_2[v]$. We can prove this proposition by induction. For j = 0, the first Root set is the same in both G_1 and G_2. Hence, it holds for j = 0. Suppose that the proposition holds for every j from 0 to k. We prove that it also holds for j= k + 1. Suppose that v is a Root of frame f_{k+1} in G_1. Then, there exists a set S reaching 2/3 of members in G_1 of frame f_k such that $\forall u \in S$ ($u \rightarrow v$). As $G_1 \sim G_2$, and v in G_2, then $\forall u \in S$ ($u \in G_2$). Because the proposition holds for j=k, as u is a Root of frame f_k in G_1, u is a Root of frame f_k in G_2. Hence, the set S of 2/3 members u occurs before v in G_2. Thus, v belongs to f_{k+1} in G_2. This proposition is proved. □

From the above proposition, one can deduce the following:

Lemma 8. G^C *is a Root, which is consistent with G_i for all nodes.*

Thus, all nodes have the same consistent Root sets, which are the Root sets in G^C. The frame numbers are consistent for all nodes.

Lemma 9 (consistent flag table)**.** *For any top event v in both consensus structures G_1 and G_2, and $G_1 \sim G_2$, then the flag tables of v are consistent iff they are the same in both structures.*

Proof. From the above lemmas, the Root sets of G_1 and G_2 are consistent. If v is contained in G_1, and v is a Root of the j-th frame in G^1, then v is a Root of the j-th frame in G_i. Because $G_1 \sim G_2$, $G_1[v] = G_2[v]$. The reference event blocks of v are the same in both chains. Thus, the flag tables of v of both chains are the same. □

Thus, all nodes have consistent flag tables.

Definition 31 (subDecision)**.** *A Root r_k in the frame f_{a+3} can nominate a Root r_a as subDecision if more than 2n/3 Roots in the frame f_{a+1} Happened-Before r_a and r_k Happened-Before the Roots in the frame f_{a+1}.*

Lemma 10. *For any Root set R, all nodes nominate the same Root into subDecision.*

Proof. Based on Theorem 2, each node nominates a Root into subDecision via the flag table. If all nodes have a consensus structure with the same shape, the values in the flag table should be equal to each other in the consensus structure. Thus, all nodes nominate the same Root into subDecision because the consensus structure of all nodes has the same shape. □

Lemma 11. *In the Reselection algorithm, for any subDecision, a Root in the consensus structure selects the same consensus time candidate.*

Proof. Based on Theorem 2, if all nodes have a consensus structure with the same partial shape, a Root in the consensus structure selects the same consensus time candidate by the Reselection algorithm. □

Lemma 12 (Fork Lemma)**.** *If the pair of event blocks (x,y) is a fork and a Root has Happened before the fork, this fork is detected in the subDecision selection process.*

Proof. We prove the lemma by contradiction. Assume that no node can detect the fork in the subDecision selection process.

Assume that there is a Root r_i that becomes a subDecision in f_i, which was selected as a subDecision by n/3 of the Roots in f_{i+1}. More than 2n/3 Roots in f_{i+1} should have occurred before by a Root in f_{i+2}. If a pair (v_x, v_y) is a fork, there are two cases: (1) assume that r_i is one of v_x and v_y, (2) assume that r_i can reach both v_x and v_y.

Our proof for both cases is as follows.

Let k denote the number of Roots in f_{i+1} that can reach r_i in f_i ($\therefore$ n/3 $< k$).

To select Root in f_{i+2}, the Root in f_{i+2} should reach more than 2n/3 of Roots in f_{i+1} by Definition 23. At the moment, assume that l is the number of Roots in f_{i+1} that can be reached by the Root in f_{i+2} ($\therefore$ 2n/3 $< l$).

At this time, n < k + l ($\because$ n/3 + 2n/3 $< k + l$), there are (n - k + l) Roots in frame f_{i+1} that should reach r_i in f_i and all Roots in f_{i+2} should reach at least n – k + l of Roots in f_{i+1}. This means that all Roots in f_{i+2} know the existence of r_i. Therefore, the node that generated all the Roots of f_{i+2} detects the fork in the subDecision selection of r_i, which contradicts the assumption.

Two cases can be covered. If r_i is part of the fork, we can detect it in f_{i+2}. If there is a fork (v_x, v_y) that can be reached by r_i, it can also be detected in f_{i+2} because we can detect the fork in the subDecision selection of r_i, which indicates that all event blocks that can be reached by r_i are detected by the Roots in f_{i+2}. □

Lemma 13. *For a Root v happened-before a fork in the consensus structure, v must see the fork before becoming a subDecision.*

Proof. Suppose that a node creates two event blocks (v_x, v_y), which form a fork. To create two subDecisions that can reach both events, it should be possible to reach the event blocks by more than 2n/3 nodes. Therefore, the consensus structure can structurally detect the fork before Roots become subDecisions. □

Theorem 2 (Fork Absence)**.** *All nodes grow into the same consistent consensus structure G^C, which contains no fork.*

Proof. Suppose that there are two event blocks v_x and v_y contained in both G_1 and G_2, and their path between v_x and v_y in G_1 are not equal to those in G_2. We can consider that the path difference between the nodes is a type of fork attack. Based on Lemma 12, if an attacker forks an event block, each chain of G_i and G_2 can detect and remove the fork before the subDecision is generated. Thus, any two nodes have a consistent consensus structure. □

Definition 32 (Decision)**.** *If the consensus time of subDecision is validated, then subDecision becomes a Decision.*

Theorem 3. *The Decision search algorithm guarantees reaching an agreement for the consensus time.*

Proof. For any Root set R in the frame f_i, the time consensus algorithm checks whether more than 2n/3 Roots in the frame f_{i-1} select the same value. However, each node selects one of the values collected from the Root set in the previous frame by the time consensus algorithm and the election process. Based on the election process, the time consensus algorithm can reach an agreement. However, the possibility exists that consensus time candidates do not reach an agreement [9]. To solve this problem, the time consensus algorithm includes a minimal selection frame for the next h frame. In the minimal value selection algorithm, each Root selects the minimum value among values collected from the previous Root set. Thus, the consensus time reaches consensus by the time consensus algorithm. □

Theorem 4. *If the number of reliable nodes is more than 2n/3, event blocks created by reliable nodes must be assigned a consensus order.*

Proof. In a consensus structure, because reliable nodes try to create event blocks by communicating with every other node continuously, reliable nodes share the event block x with each other. If a Root y in the frame f_i Happened-Before event block x and more than n/3 Roots in the frame f_{i+1} Happened-Before the Root y, the Root y is nominated as subDecision and Decision. Thus, event blocks x and Root y are assigned a consensus time t.

For an event block, assigning consensus time means that the validated event block is shared by more than 2n/3 nodes. Therefore, a malicious node cannot try to attack after the event blocks are assigned a consensus time. When the event block x has consensus time t, it cannot occur to discover new event blocks with an earlier consensus time than t. Two conditions are to be assigned a consensus time earlier than t for new event blocks. First, a Root r in frame f_i should be able to share new event blocks. Second, more than 2n/3 Roots in the frame f_{i+1} should be able to share r. Even if the first condition was satisfied by malicious nodes (e.g., a parasite chain), the second condition cannot be satisfied because at least 2n/3 Roots in the frame f_{i+1} are already created and cannot be changed. Therefore, after an event block is validated, new event blocks should not participate earlier in consensus on the consensus structure. □

4.2. Semantics of Decision Search Protocol

This section provides the formal semantics of the Decision search protocol. We use the CCK model [24] of an asynchronous system as the base of the semantics of our Decision search protocol. Events are ordered based on Lamport's happens-before relation. In particular, we use Lamport's theory to describe the global states of an asynchronous system.

We present notations and concepts that are important for the Decision search protocol. In several places, we adapt the notations and concepts used in the paper in which CCK was proposed to suit our Decision search protocol.

An asynchronous system consists of the following:

Definition 33 (process)**.** *Process p_i represents a machine or a node. The process identifier of p_i is i. A set P = {1,...,n} denotes the set of process identifiers.*

Definition 34 (channel)**.** *Process i can send messages to process j if there is a channel (i,j). Let C* $\subseteq$ *{(i,j) s.t.* $i, j \in P$*} denote the set of channels.*

Definition 35 (state)**.** *The local state of a process i is denoted by* s^i_j*.*

A local state consists of a sequence of event blocks, $s^i_j = v^i_0, v^i_1, \ldots, v^i_j$.

In a DAG-based protocol, each v^i_j event block is valid only when the reference blocks exist before it. A unique DAG can be reconstructed from a local state s^i_j. That is, the mapping from a local state s^i_j into a DAG is injective or one-to-one. Thus, for the Decision search, we can simply denote the *j*-th local state of a process *i* by the consensus structure g^i_j (often we simply use G_i to denote the current local state of a process *i*).

Definition 36 (action)**.** *An action is a function from one local state to another local state.*

Typically, an action can be a $send(m)$ action, where m is a message, a $receive(m)$ action, or an internal action. A message m is a triple $\langle i, j, B\rangle$, where $i \in P$ is the sender of the message, $j \in P$ is the message recipient, and B is the body of the message. Let M denote the set of messages. In the Decision search protocol, B consists of the content of an event block v. Semantics-wise, in Lachesis, two actions can change the local state of a process: creating a new event and receiving an event from another process.

Definition 37 (event)**.** *An event is a tuple* $\langle s, \alpha, s'\rangle$ *consisting of a state, an action, and a state.*

Under certain circumstances, the event can be represented by the end state s'. The *j*-th event in history h_i of process *i* is $\langle s^i_{j-1}, \alpha, s^i_j\rangle$, denoted by v^i_j.

Definition 38 (local history)**.** *A local history* h_i *of a process i is a (possibly infinite) sequence of alternating local states—beginning with a distinguished initial state. A set* H_i *of the possible local histories for each process i in P.*

The state of a process can be obtained from its initial state and the sequence of actions or events that have occurred up to the current state. In the Decision search protocol, we use append-only semantics. The local history may be equivalently described as one of the following:

$$h_i = s^i_0, \alpha^i_1, \alpha^i_2, \alpha^i_3 \ldots$$

$$h_i = s^i_0, v^i_1, v^i_2, v^i_3 \ldots$$

$$h_i = s^i_0, s^i_1, s^i_2, s^i_3, \ldots$$

In the Decision search, a local history is equivalently expressed as

$$h_i = g^i_0, g^i_1, g^i_2, g^i_3, \ldots$$

where g^i_j is the *j*-th local consensus structure (local state) of process *i*.

Definition 39 (run)**.** *Each asynchronous run is a vector of local histories. Denoted by* $\sigma = \langle h_1, h_2, h_3, ...h_N\rangle$*.*

Let Σ denote the set of asynchronous runs.

We can now use Lamport's theory to discuss the global states of an asynchronous system. A global state of run σ is an n-vector of prefixes of local histories of σ, one prefix per process. The happens-before relation can be used to define a consistent global state, often termed a consistent cut, as follows.

Definition 40 (Consistent cut)**.** *A consistent cut of a run σ is any global state such that if $v^i_x \to v^j_y$ and v^j_y is in the global state, then v^i_x is also in the global state. We denote $\mathbf{c}(\sigma)$.*

According to Theorem 1, all nodes have consistent local consensus structures. The concept of a consistent cut formalizes the global state of a run. A consistent cut consists of all consistent consensus structures. A received event block exists in the global state, implying the existence of the original event block. Note that a consistent cut is simply a vector of local states; we use the notation $\mathbf{c}(\sigma)[i]$ to indicate the local state of i in cut $\mathbf{c}$ of run σ.

The formal semantics of an asynchronous system are given via the satisfaction relation $\vdash$. Intuitively, $\mathbf{c}(\sigma) \vdash \phi$, "$\mathbf{c}(\sigma)$ satisfies ϕ," if fact ϕ is true in cut $\mathbf{c}$ of run σ. We assume that we are given a function π that assigns a truth value to each primitive proposition p. The truth of a primitive proposition p in $\mathbf{c}(\sigma)$ is determined by π and $\mathbf{c}$. This defines $\mathbf{c}(\sigma) \vdash p$.

Definition 41 (equivalent cuts)**.** *Two cuts $\mathbf{c}'(\sigma')$ and $\mathbf{c}'(\sigma')$ are equivalent to i if*

$$\mathbf{c}'(\sigma') \sim_i \mathbf{c}'(\sigma') \Leftrightarrow \mathbf{c}'(\sigma')[i] = \mathbf{c}'(\sigma')[i]$$

We introduce two families of modal operators, denoted by K_i and P_i, respectively. Each family is indexed by process identifiers. Given a fact ϕ, the modal operators are defined as follows:

Definition 42 (i knows ϕ)**.** *$K_i(\phi)$ represents the statement "ϕ is true in all possible consistent global states that include the local state of i."*

$$\mathbf{c}'(\sigma') \vdash K_i(\phi) \Leftrightarrow \forall \mathbf{c}'(\sigma')(\mathbf{c}'(\sigma') \sim_i \mathbf{c}'(\sigma') \;\Rightarrow\; \mathbf{c}'(\sigma') \vdash \phi),$$

Definition 43 (i partially knows ϕ)**.** *$P_i(\phi)$ represents the statement "there is some consistent global state in this run that includes the local state of i, in which ϕ is true."*

$$\mathbf{c}(\sigma) \vdash P_i(\phi) \Leftrightarrow \exists \mathbf{c}'(\sigma)(\mathbf{c}'(\sigma) \sim_i \mathbf{c}(\sigma) \;\wedge\; \mathbf{c}'(\sigma) \vdash \phi),$$

The next modal operator is written as M^C and stands for "majority concurrently knows." This is adapted from the "everyone concurrently knows" in the CCK paper [24]. The definition of $M^C(\phi)$ is as follows:

Definition 44 (majority concurrently knows)**.**

$$M^C(\phi) =_{def} \bigwedge_{i \in S} K_i P_i(\phi),$$

where $S \subseteq P$ and $|S| > 2n/3$.

In the presence of one-third of faulty nodes, the original operator "everyone concurrently knows" is not necessarily feasible. Our modal operator $M^C(\phi)$ precisely fits the semantics for BFT systems, in which unreliable processes may exist.

The last modal operator is concurrent common knowledge (CCK), denoted by C^C.

Definition 45 (concurrent common knowledge)**.** *$C^C(\phi)$ is defined as a fixed point of $M^C(\phi \wedge X)$,*

CCK defines a state of process knowledge that implies that all processes are in the same state of knowledge, with respect to ϕ, along a certain cut of the run. In other words, we aim to achieve a state of knowledge X satisfying $X = M^C(\phi \wedge X)$. C^C will be defined semantically as the weakest fixed point, namely, the greatest fixed-point of $M^C(\phi \wedge X)$. It therefore satisfies:

$$C^C(\phi) \Leftrightarrow M^C(\phi \wedge C^C(\phi)),$$

Thus, $P_i(\phi)$ states that there is a certain cut in the same asynchronous run σ, including the local state of i, such that ϕ is true in that cut.

Note that ϕ implies $P_i(\phi)$. However, this is not the case, in general, that $P_i(\phi)$ implies ϕ, or even that $M^C(\phi)$ implies ϕ. The truth of $M^C(\phi)$ is determined with respect to a certain cut $\mathbf{c}(\sigma)$. A process cannot distinguish which cut of the perhaps many cuts that are in the run and are consistent with its local state, satisfies ϕ; it only knows that such a cut exists.

Definition 46 (global fact)**.** *Fact ϕ is valid in system Σ, denoted by $\Sigma \vdash \phi$, if ϕ is true in all cuts of all runs of Σ.*

$$\Sigma \vdash \phi, \Leftrightarrow (\forall \sigma \in \Sigma)(\forall \mathbf{c})(\mathbf{c}(a) \vdash \phi)$$

Definition 47. *Fact ϕ is valid, denoted by $\vdash \phi$, if ϕ is valid in all systems, that is, $(\forall \Sigma)(\Sigma \vdash \phi)$,*

Definition 48 (local fact)**.** *The fact that ϕ is local to process i in system Σ if $\Sigma \vdash (\phi \Rightarrow K_i\phi)$,*

Theorem 5. *If ϕ is local to process i in system Σ, then $\Sigma \vdash (P_i(\phi) \Rightarrow \phi)$.*

Lemma 14. *If ϕ is local to 2/3 of the processes in a system Σ, then $\Sigma \vdash (M^C(\phi) \Rightarrow \phi)$ and $\Sigma \vdash (C^C(\phi) \Rightarrow \phi)$.*

Definition 49. *Indeed, ϕ is attained in run σ if $\exists \mathbf{c}(\sigma)(\mathbf{c}(\sigma) \vdash \phi)$.*

Often, we refer to "knowing" a fact ϕ in a state rather than in a consistent cut, because knowledge is dependent only on the local state of a process. Formally, i knows ϕ in state s is shorthand for

$$\forall \mathbf{c}(\sigma)(\mathbf{c}(\sigma)[i] = s \Rightarrow \mathbf{c}(\sigma) \vdash \phi),$$

For example, if a process in the Decision search protocol knows that a fork exists (i.e., ϕ is the existence of fork) in its local state s (i.e., g^i_j), then a consistent cut that contains the state s will know the existence of that fork.

Definition 50 (i learns ϕ)**.** *Process i learns ϕ in state s^i_j of run σ if i knows ϕ in s^i_j and, for all previous states s^i_k in run σ, $k < j$, i does not know ϕ.*

The following theorem states that if $C_C(\phi$ is attained in a run, then all processes i learn $P_iC^C(\phi)$ along a single consistent cut.

Theorem 6 (attainment)**.** *If $C^C(\phi)$ is attained in a run σ, then the set of states in which all processes learn $P_iC^C(\phi)$ forms a consistent cut in σ.*

We presented the formal semantics of the Decision search protocol based on the concepts and notations of CCK [24]. The proofs of the theorems and lemmas in this Section are similar to those in the original CCK paper.

With the formal semantics of the Decision search, the theorems and lemmas described in Section 4 can be expressed in terms of CCK. For example, a fact ϕ (or some primitive proposition p) can be studied in the following forms: "does a fork exist?" Decision search-specific questions such as "is event block v a Root?", "is v a subDecision?", or "is v a Decision" could be formulated. This is a remarkable result, because we are the first to define such formal semantics for a DAG-based protocol.

5. Decision Search Protocol DecisionBFT

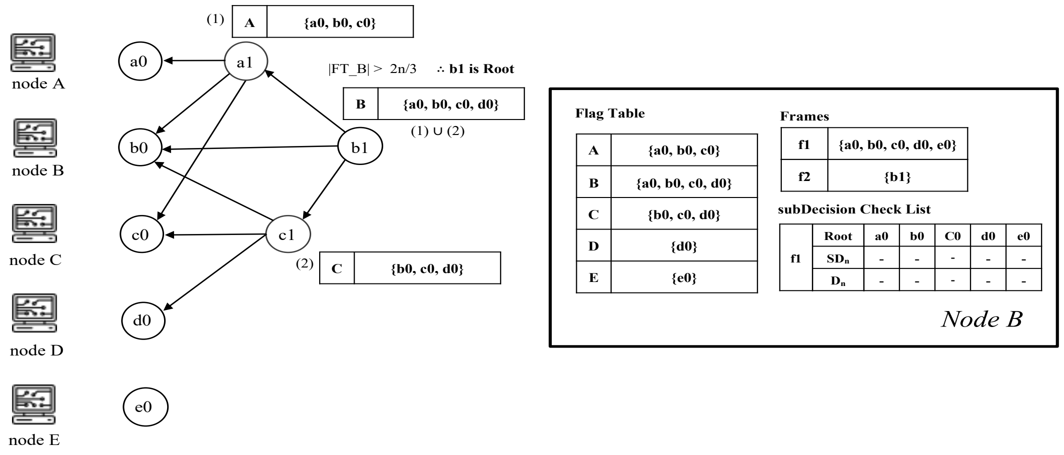
This section presents our new Decision Search Protocol DecisionBFT, which is a specific example of the Decision Search class. We describe the main ideas and algorithms used in the protocol.

5.1. Root Selection

All nodes can create event blocks, and an event block can be a Root when it satisfies specific conditions. Not all event blocks can be Roots. First, the first created event blocks are Roots. These Leaf event blocks form the first Root set R_{S_1} of the first frame f_1. If there are n nodes and these nodes create the event blocks, then the cardinality of the first Root set $|R_{S_1}|$ is n. Second, if an event block e can reach at least 2n/3 Roots, then e is referred to as a Root. This event e does not belong to R_{S1}, but the next Root set R_{S_2} of the next frame f_2. Thus, excluding the first Root set, the range of cardinality of the Root set R_{S_k} is $2n/3 < |R_{S_k}| \leq n$. The event blocks including R_{S_k} before $R_{S_{k+1}}$, are in the frame f_k. The Roots in $R_{S_{k+1}}$ do not belong to frame f_k. These are included in the frame f_{k+1} when a Root belonging to $R_{S_{k+2}}$ occurs.

We introduce the use of a flag table to quickly determine whether a new event block becomes a Root. Each node maintains a flag table of the top event block. Every event block that is newly created is assigned k hashes for its k reference event blocks. We apply an *OR* operation to each set in the flag tables of the reference event blocks.

Figure 12 shows an example of the use of flag tables to determine a Root. In this example, b_1 is the most recently-created event block. We apply an *OR* operation to each set of the flag tables for the k reference event blocks of b_1. The result is the flag table of b_1. If the cardinality of the Root set in the flag table of b_1 is more than 2n/3, b_1 is a Root. In this example, the cardinality of the Root set in b_1 is 4, which is greater than 2n/3 (n = 5). Thus, b_1 becomes a Root. In this example, b_1 is added to frame f_2 because b_1 becomes the new Root.



Flag Table

A	{a0, b0, c0}
B	{a0, b0, c0, d0}
C	{b0, c0, d0}
D	{d0}
E	{e0}

Frames

f1	{a0, b0, c0, d0, e0}
f2	{b1}

subDecision Check List

	Root	a0	b0	C0	d0	e0
f1	SD_n	-	-	-	-	-
	D_n	-	-	-	-	-

Figure 12. Example of Root selection.

The Root selection algorithm is as follows:

1. The first event blocks are considered as Roots.
2. When a new event block is added to the consensus structure, we check whether the event block is a Root by applying an *OR* operation to each set of the flag tables connected to the new event block. If the cardinality of the Root set in the flag table for the new event block is more than 2n/3, the new event block becomes a Root.
3. When a new Root appears on the consensus structure, the nodes update their frames. If one of the new event blocks becomes a Root, all nodes that share the new event block add the hash value of the event block to their frames.

4. The new Root set is created if the cardinality of the previous Root set R_{S_p} is more than 2n/3 and the new event block can reach 2n/3 Root in R_{S_p}.
5. When the new Root set $R_{S_{k+1}}$ is created, the event blocks from previous Root set R_{S_k} to before $R_{S_{k+1}}$ belong to the frame f_k.

5.2. subDecision Selection

A subDecision is a Root that satisfies the conditions under which the subDecision is created. These conditions are that more than 2n/3 nodes know the Root and a Root knows this information.

For a Root r in frame f_i to become a subDecision, r must be reached by more than n/3 Roots in the frame f_{i+1}. Based on the definition of the Root, each Root reaches more than 2n/3 Roots in previous frames. If more than n/3 Roots in the frame f_{i+1} can reach r, then r is spread to all Roots in the frame f_{i+2}. This means that all nodes know the existence of r. If we have any Roots in the frame f_{i+3}, a Root knows that r is spread to more than 2n/3 nodes. It satisfies the subDecision creation conditions.

In the example in Figure 13, n is 5, and each circle indicates a Root in a frame. Each arrow indicates that one Root can reach (happened-before) to the previous Root. Each Root has four or five arrows (out-degree) because n is 5 (more than $2n/3 \geq 4$). b_1 and c_2 in frame f_2 are Roots that can reach c_0 in frame f_1. d_1 and e_1 can also reach c_0, but we only marked b_1 and c_2 (when n is 5, more than $n/3 \geq 2$) because we require at least n/3 conditions in this example. This is marked with a blue bold arrow (i.e., the Roots that can reach Root c_0 are indicated by a blue bold arrow). In this situation, an event block must be able to reach b_1 or c_2 to become a Root in frame f_3 (in our example, n = 5, more than $n/3 \geq 2$, and more than $2n/3 \geq 4$. To be a Root, either must be reached). Therefore, all Roots in frame f_3 reach c_0 in frame f_1.

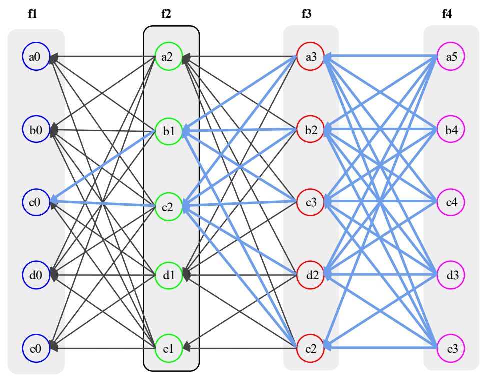


Figure 13. Verification of more than 2n/3 nodes.

To be a Root in frame f_4, an event block must reach more than 2n/3 Roots in frame f_3 that can reach c_0. Therefore, if any of the Root in frame f_4 exists, then Root must be happened-before more than 2n/3 Roots in frame f_3. This means that the Root of f_4 knows that c_0 is spread over more than 2n/3 of the entire node. Thus, we can select c_0 as a subDecision.

Figure 14 shows an example of a subDecision. In this example, all Roots in the frame f_1 are happened-before by more than n/3 Roots in the frame f_2. We can select all Roots in the frame f_1 as a subDecision because the most recent frame is f_4.

Figure 15 shows the state of node A when a subDecision is selected. In this example, node A knows all Roots in the frame f_1 become a subDecision.

Algorithm 3 shows the pseudocode for subDecision selection. The algorithm takes Root r as input. Lines 4 and 5 set $c.is\_subDecision$ and $c.yes$ to nil and 0, respectively. Lines 6–8 check whether any Root c' in $frame(i-3,r)$ has occurred before with the 2n/3 condition c, where i is the current frame. In lines 9–10, if the number of Roots in $frame(i-2,r)$ that occurs before c is more than 2n/3, then the Root c is set as a subDecision. The time complexity of Algorithm 3 is $O(n^2)$, where n is the number of nodes.

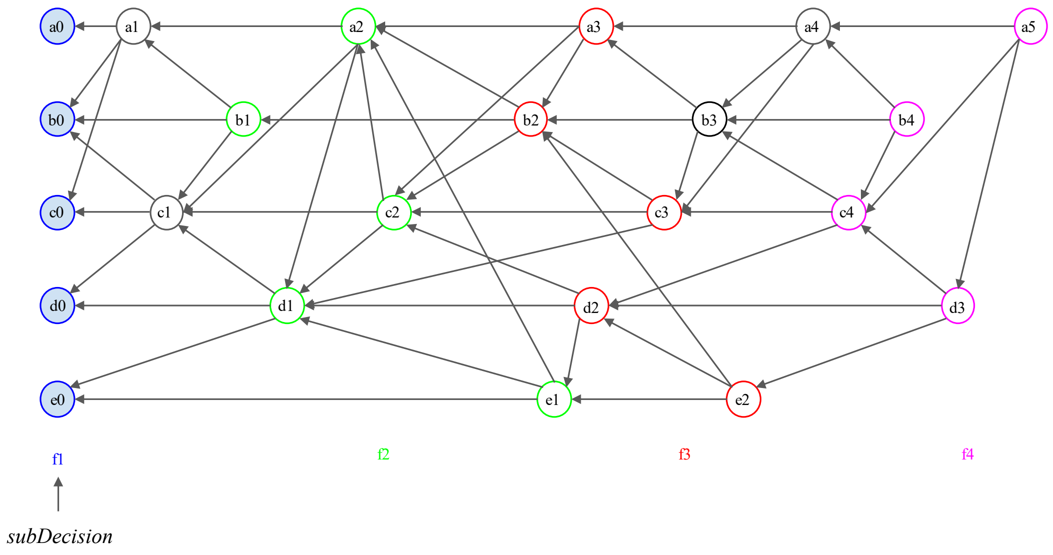
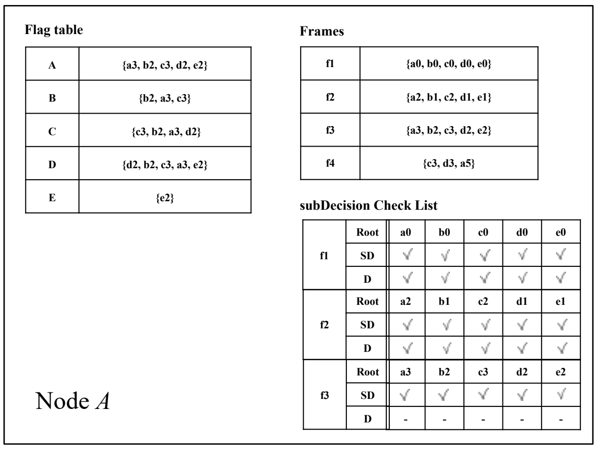


Figure 14. Example of a subDecision.



Flag table

A	{a3, b2, c3, d2, e2}
B	{b2, a3, c3}
C	{c3, b2, a3, d2}
D	{d2, b2, c3, a3, e2}
E	{e2}

Frames

f1	{a0, b0, c0, d0, e0}
f2	{a2, b1, c2, d1, e1}
f3	{a3, b2, c3, d2, e2}
f4	{c3, d3, a5}

subDecision Check List

f1	Root	a0	b0	c0	d0	e0
	SD	✓	✓	✓	✓	✓
	D	✓	✓	✓	✓	✓
f2	Root	a2	b1	c2	d1	e1
	SD	✓	✓	✓	✓	✓
	D	✓	✓	✓	✓	✓
f3	Root	a3	b2	c3	d2	e2
	SD	✓	✓	✓	✓	✓
	D	-	-	-	-	-

Node *A*

Figure 15. Node of *A* when subDecision is selected.

Algorithm 3 The subDecision selection

```
1:  procedure SUBDECISION SELECTION
2:      Input: a Root $r$
3:      for $c \in frame(i-3, r)$ do
4:          $c.is\_subDecision \leftarrow nil$
5:          $c.yes \leftarrow 0$
6:          for $c' \in frame(i-2, r)$ do
7:              if $c'$ has happened-before $c$ then
8:                  c.yes ← c.yes + 1
9:          if $c.yes > 2n/3$ then
10:             $c.is\_subDecision \leftarrow yes$
```

A subDecision can know to spread to more than 2n/3 nodes. It is used for Decision decisions. Our algorithm for Decision determines the consensus time for a subDecision rather than all event blocks. A subDecision is therefore necessary for the process of finality for the BFT.

5.3. Decision Selection

The Decision selection algorithm is the process in which the candidate time generated from a subDecision selection is shared with other nodes, and each Root re-selects the candidate time repeatedly until all nodes have the same candidate time for a subDecision.

After a subDecision is nominated, each node then computes the candidate time of the subDecision. If more than two-thirds of the nodes compute the same value for candidate time, that time value is recorded. Otherwise, each node re-selects the candidate time from some candidate time that the node collects. By the reselection process, each node reaches a time consensus for the candidate time of the subDecision as the consensus structure grows. The candidate time reaching the consensus is referred to as the Decision consensus time. After the Decision consensus time is computed, a subDecision is nominated to Decision, and each node stores the hash value of a Decision and the Decision consensus time. The proof of the Decision consensus time selection is presented in Section 4.

Figure 16 shows an example of a Decision. In this Figure 14, all Roots in the frame f_1 are selected as a subDecision through the existence of Roots in the frame f_4. Then, each Root in the frame f_5 computes the candidate time using the timestamps of reachable Roots in the frame f_4. Each Root in the frame f_5 stores the candidate time to min-max value space. The Root r_6 in the frame f_6 can reach more than 2n/3 Roots in f_5, and r_6 can know the candidate time the reachable Roots in f_5 takes. If r_6 knows the same candidate time more than 2n/3, we select the candidate time as Decision consensus time. Then all subDecisions in the frame f_1 become Decisions.

Figure 17 shows the state of node B when a Decision is selected. In this example, node B knows all Roots in the frame f_1 become Decisions.

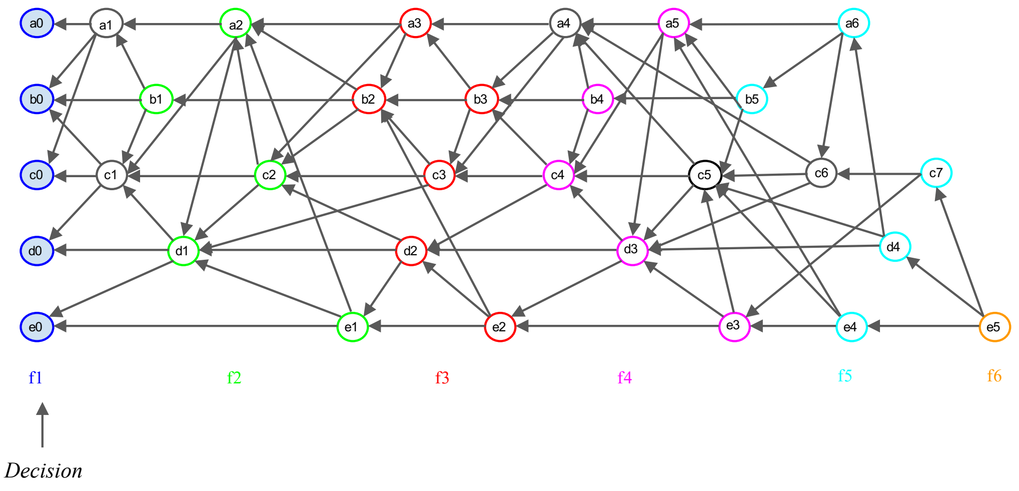
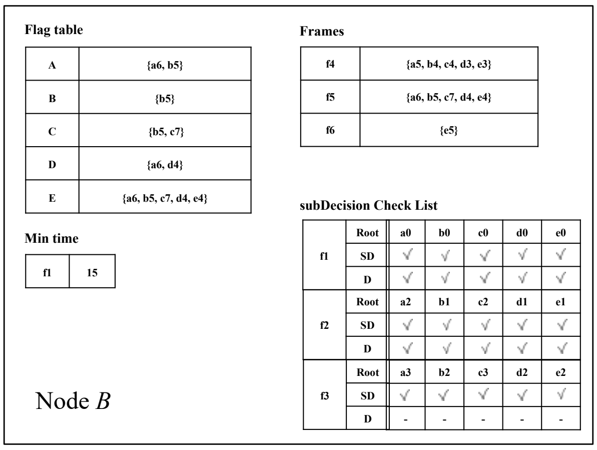


Figure 16. Example of a Decision.

Algorithms 4 and 5 show the pseudocode for the Decision consensus time selection and consensus time re-selection. In Algorithm 4, at line 6, d saves the difference in the relationship between the Root set of c and w. Thus, line 8 means that w is one of the elements in the Root set of the frame f_{i+3}, where the frame f_i includes c. In line 10, each Root in the frame f_j selects its own Lamport timestamp as a candidate time of c when they confirm Root c as a subDecision. In lines 14, 15, and 16, s, t, and k save the set of Root that w can be happened-before with 2n/3 condition c, the result of the RE-SELECTION function, and the number of Roots in s with t. Line 17 checks whether a difference of as much as h exists between i and j, where h is a constant value for the minimum selection frame. Lines 18–22 check whether more than two-thirds of Root in the frame f_{j-1} nominates the same candidate time. If two-thirds of Roots in the frame f_{j-1} nominates the same candidate time, the Root c is assigned a consensus time t. Line 24 is the minimum selection frame, in which the minimum value of the candidate time is selected to reach the byzantine agreement. Algorithm 5 is executed within

Algorithm 4. The input of Algorithm 5 is the Root set W, and the output is the re-selected candidate time. Lines 4–5 compute the frequencies of each candidate time from all the Roots in W. In lines 6–15, the candidate time that is the smallest time is the most nominated. The time complexity of Algorithm 5 is $O(n)$, where n is the number of nodes. Because Algorithm 4 includes Algorithm 5, the time complexity of Algorithm 4 is $O(n^2)$, where n is the number of nodes.



Flag table

A	{a6, b5}
B	{b5}
C	{b5, c7}
D	{a6, d4}
E	{a6, b5, c7, d4, e4}

Frames

f4	{a5, b4, c4, d3, e3}
f5	{a6, b5, c7, d4, e4}
f6	{e5}

Min time

f1	15

subDecision Check List

f1	Root	a0	b0	c0	d0	e0
	SD	✓	✓	✓	✓	✓
	D	✓	✓	✓	✓	✓
f2	Root	a2	b1	c2	d1	e1
	SD	✓	✓	✓	✓	✓
	D	✓	✓	✓	✓	✓
f3	Root	a3	b2	c3	d2	e2
	SD	✓	✓	✓	✓	✓
	D	-	-	-	-	-

Node B

Figure 17. Node of B when a Decision is selected.

Algorithm 4 Decision consensus time selection

```
1: procedure DECISION CONSENSUS TIME SELECTION
2:     Input: $c$: subDecision in frame $f_i$
3:     $c.consensus\_time \leftarrow nil$
4:     $m \leftarrow$ the index of the last frame $f_m$
5:     for d from 3 to (m-i) do
6:         $R \leftarrow$ be the Root set $R_{S_{i+d}}$ in frame $f_{i+d}$
7:         for $r \in R$ do
8:             if d is 3 then
9:                 if $r$ confirms $c$ as subDecision then
10:                    $r.time(c) \leftarrow r.lamport\_time$
11:                else
12:                    $r.time(c) \leftarrow INF$
13:            else if d $>$ 3 then
14:                s $\leftarrow$ the set of Root in $f_{m-1}$ that $r$ can have happened-before with 2n/3 condition
15:                t $\leftarrow$ RE-SELECTION(s, $c$)
16:                k $\leftarrow$ the number of Root having $t$ in $s$
17:                if d mod $h > 0$ then
18:                    if $k >$ 2n/3 then
19:                        $c.consensus\_time \leftarrow t$
20:                        $r.time(c) \leftarrow t$
21:                    else
22:                        $r.time(c) \leftarrow t$
23:                else
24:                    $r.time(c) \leftarrow$ the minimum value in $s$
```

Algorithm 5 Consensus Time Re-selection

```
 1: function RE-SELECTION
 2:     Input: Root set $R$, and subDecision $c$
 3:     Output: candidate time $t$
 4:     $\tau \leftarrow$ set of all $t_i = r.time(c)$ for all $r$ in $R$
 5:     $D \leftarrow$ set of tuples $(t_i, c_i)$ computed from $\tau$, where $c_i = count(t_i)$
 6:     $max\_count \leftarrow max(c_i)$
 7:     $t \leftarrow infinite$
 8:     for tuple $(t_i, c_i) \in D$ do
 9:         if $max\_count == c_i$ then
10:             if $t_i == INF$ && $c_i > |R|/2$ then
11:                 $returnINF$
12:             else if $t_i < t$ then
13:                 $t = t_i$
14:             $t \leftarrow t_i$
15:     return $t$
```

In the Decision Consensus Time Selection algorithm, nodes reach consensus agreement about the candidate time of a subDecision without additional communication (i.e., exchanging candidate time) with each other. Each node communicates with each other through the Decision search protocol, and the consensus structure of all nodes grows into the same shape. This allows each node to know the candidate time of other nodes and reach a consensus agreement. The proof that the agreement based on the consensus structure becomes agreement in action is shown in Section 4.

A Decision can be determined by the consensus time of each subDecision. It is an event block that is determined by finality and is non-modifiable. Furthermore, all event blocks can be reached from the Decision guarantee finality.

5.4. Peer Selection Algorithm via Cost Function

We define three versions of the cost function (C_F). The version on which we focus is the updated information share and is discussed below. The other two versions focus on Root creation and consensus facilitation, which will be discussed in a subsequent paper.

We define a cost function (C_F) to prevent the creation of lazy nodes, which is a node that performs a smaller portion of work. When a node creates an event block, the node selects other nodes with low values as outputs by using the cost function and refers to the top event blocks of the reference nodes. Equation (1) of C_F is as follows:

$$C_F = I/H, \tag{1}$$

where I and H denote the values of the in-degree vector and height vector, respectively. If the number of nodes with the lowest C_F exceeds k, one of the nodes is selected at random. The reason for selecting a large value of H is that we can expect a high possibility to create a Root because the large H indicates that the communication frequency of the node had more opportunities than others with a small H. Otherwise, the nodes with large C_F (the case of $I > H$) generated fewer event blocks than the nodes with small C_F. Then, we can judge that those kinds of nodes are lazy. If we can detect whether a node is lazy based on the cost function, we can change the lazy nodes to other participants or remove them.

Figure 18 shows an example of the node selection based on the cost function after the creation of Leaf events by all nodes. This example is based on five nodes and each node created Leaf events. All nodes know other Leaf events. Node A creates an event block v_1 and A calculates the cost functions.

Step 2 in Figure 18 shows the results of cost functions based on the height and in-degree vectors of node A. In the initial step, each value in the vectors is the same because all nodes have only leaf events. Node A randomly selects k nodes and connects v_1 to the Leaf events of selected nodes. In this example, we set k = 3 and assume that node A selects node B and C.

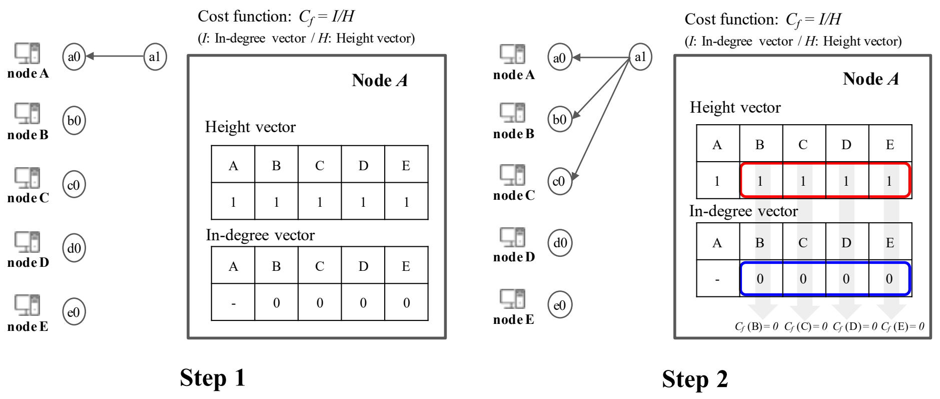


Figure 18. Example of a cost function 1.

Figure 19 shows an example of node selection after a few steps of the simulation in Figure 18. In Figure 19, the most recent event block is v_5 created by node A. Node A calculates the cost function and selects the other two nodes that have the lowest results of the cost function. In this example, node B has 0.5 as the result and the other nodes have the same values. Because of this, node A first selects node B and randomly selects the other nodes from among nodes C, D, and E.

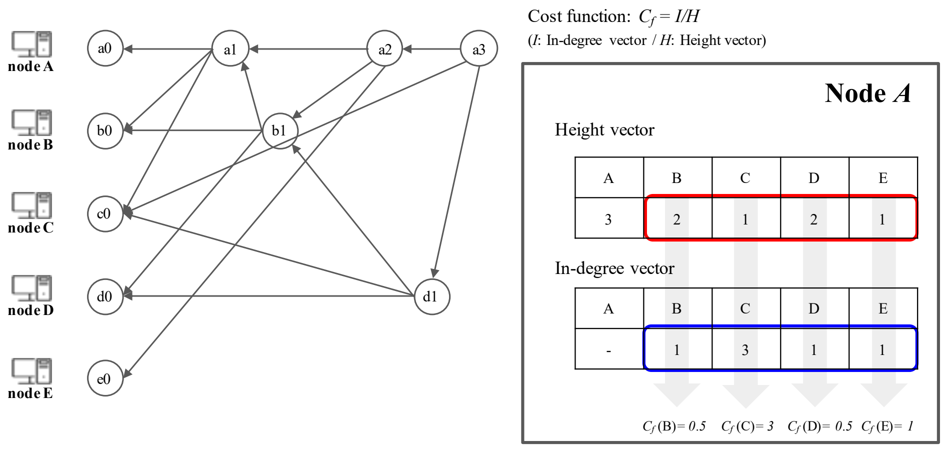


Figure 19. Example of a cost function 2.

The height of node D in the current consensus structure of the example is 2 (Leaf event and event block v_4). On the other hand, the height of node D in the node structure of A is 1. Node A is still not aware of the presence of event block v_4. This means that a path does not exist from the event blocks created by node A to event block v_4. Thus, node A has 1 as the height of node D.

Algorithm 2 contains the pseudocode for selecting reference nodes. The algorithm is executed to enable each node to select a communication partner from among the other nodes. Lines 4 and 5 set min_cost and S_{ref} to the initial state. Line 7 calculates the cost function c_f for each node. In lines 8–10, we find the minimum value of the cost function and set min_cost and S_{ref} to c_f and the ID of each node, respectively. Lines 11 and 12 append the ID of each node to S_{ref} if min_cost equals c_f. Finally, line 13 randomly selects k node IDs from S_{ref} as communication partners. The time complexity of Algorithm 2 is $O(n)$, where n is the number of nodes.

After the reference node is selected, each node communicates and shares information about all the event blocks known by them. A node creates an event block by referring to the top event block of the reference node. The Decision search protocol works and communicates asynchronously. This allows a node to create an event block asynchronously even when another node creates an event block. Communication between nodes does not allow simultaneous communication with the same node.

Figure 20 shows an example of the node selection in the Decision search protocol. This example is based on five nodes (A, B, C, D, and E) and each node generates the first event blocks, referred to as Leaf events. All nodes share other Leaf events with each other. In the first step, node A generates new event block a_1. Then, node A calculates the cost function to connect other nodes. In this initial situation, all nodes have one event block in the form of a leaf event, thus the height vector and the in-degree vector in node A have the same values. In other words, the heights of each node are 1 and the values of the in-degree vectors are 0. Thus, node A randomly selects two other nodes and connects a_1 to the top two event blocks by two other nodes. Step 2 shows the situation after connections. In this example, node A selects nodes B and C to connect a_1 and the event block a_1 is connected to the top event blocks of node B and C. Node A only knows the situation after step 2.

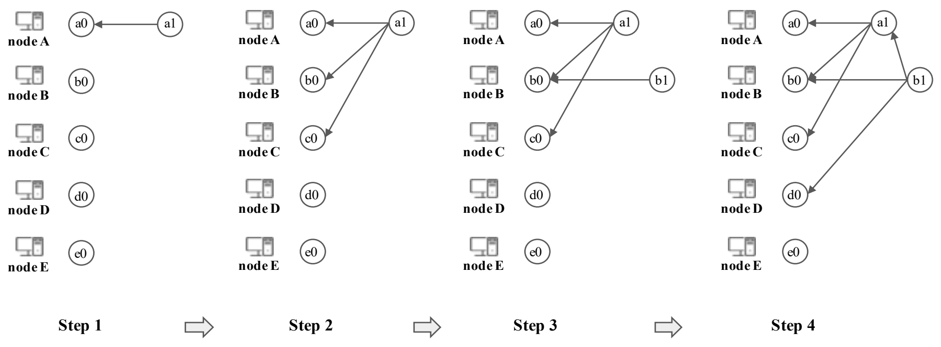
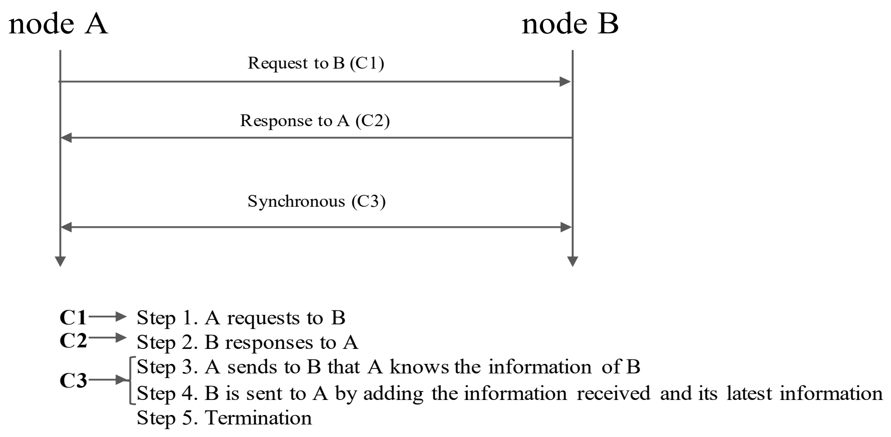


Figure 20. Example of node selection.

After that, in the example, node B generates a new event block b_1 and calculates the cost function. B randomly selects the other two nodes, A and D because B only has information about the leaf events. Node B requests A and D to connect b_1, then nodes A and D send information for their top event blocks to node B as a response. The top event block of node A is a_1, and node D is the Leaf event. The event block b_1 is connected to a_1 and Leaf event from node D. Step 4 shows these connections.

In Figure 21, the communication process is divided into five steps for two nodes to create an event block. Simply, node A submits requests to B, in which case B responds directly to A. When two nodes can interact with each other, the steps they use to share their latest information (three steps) are synchronized.


node A
node B
Request to B (C1)
Response to A (C2)
Synchronous (C3)
C1 → Step 1. A requests to B
C2 → Step 2. B responses to A
C3 → Step 3. A sends to B that A knows the information of B
Step 4. B is sent to A by adding the information received and its latest information
Step 5. Termination
Figure 21. Example of a communication process.

6. Discussions

This section presents several discussions on our Decision search protocol.

6.1. Lamport Timestamps

This section discusses the topological order of event blocks in Decision search protocols using Lamport timestamps [23].

Our protocols rely on Lamport timestamps to define the topological ordering of event blocks. The use of these timestamps makes it unnecessary to rely on physical locks to determine the partial ordering of events.

The "happened-before" relation, denoted by $\rightarrow$, gives a partial ordering of events from a distributed system of nodes. Each node n_i (also termed a process) is identified by its process identifier i. For a pair of event blocks v and v', the relation "$\rightarrow$" satisfies (1) if v and v' are events of process P_i, and v precedes v', then $b \rightarrow v'$. (2) If v is the send(m) by one process and v' is the receive(m) by another process, then $v \rightarrow v'$. (3) If $v \rightarrow v'$ and $v' \rightarrow v''$ then $v \rightarrow v''$. Two distinct events v and v' are said to be concurrent if $v \nrightarrow v'$ and $v' \nrightarrow v$.

For an arbitrary total ordering $\prec$ of the processes, the relation $\Rightarrow$ is defined as follows: if v is an event in process P_i and v' is an event in process P_j, then $v \Rightarrow v'$ if and only if either (i) $C_i(v) < C_j(v')$ or (ii) $C(v) = C_j(v')$ and $P_i \prec P_j$. This defines a total ordering, and the Clock Condition implies that if $v \rightarrow v'$, then $v \Rightarrow v'$.

We use this total ordering in our DSA. This ordering is used to determine the consensus time, as described in Sections 3 and 5.

6.2. Semantics of Decision Search Protocols

This section discusses the possible usage of concurrent common knowledge, described in Section 2.1.4, to understand DAG-based consensus protocols.

In the Decision search protocol, the consensus structure $G = (V, E)$ is a DAG (i.e., a directed graph with no cycles), where V is a set of vertices and E is a set of edges. A path with its source and destination at the same vertex does not exist. A path is a sequence of vertices (v_1, v_2, ..., v_{k-1}, v_k) that uses no edge more than once.

An asynchronous system consists of the following sets: a set P of process identifiers, a set C of channels, a set H_i of possible local histories for each process i, a set A of asynchronous runs, and a set M of all messages.

Each process of a node selects k other nodes as peers. For a certain gossip protocol, nodes may be constrained to gossip with their k peers. In such a case, the set of channels C can be modeled as follows: If node i selects node j as a peer, then $(i, j) \in C$. In general, one can express the history of each node in the DAG-based protocol in general or in the gossip protocol. This is discussed in the paper that focused on CCK [24].

The proof and formal semantics of Lachesis are described in Section 4.

6.3. Dynamic Participants

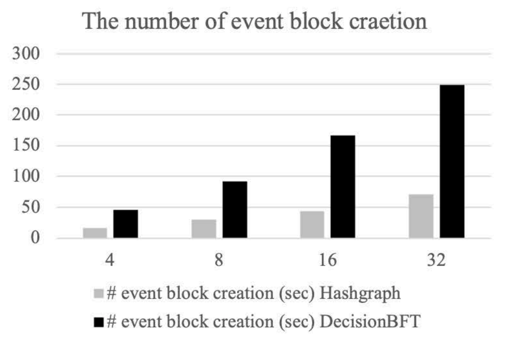
Our Decision search protocol allows an arbitrary number of participants to dynamically join the system. The operation of the consensus structure would still be possible with new participants. Computation on the flag tables is set based and does not depend on the nature and number of participants that joined the system. The algorithms for the selection of Roots, subDecision, and Decision are sufficiently flexible and are not dependent on a fixed number of participants.

6.4. Analysis of Time Complexity

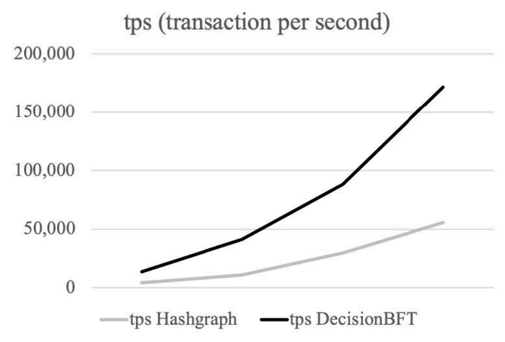
For the existing protocol [17] utilizing DAG-based gossip protocols for block-sharing detection, for n participating nodes, it is necessary to check whether the block can reach 2n/3 nodes when a new block occurs. This is based on rounds, and as the number of rounds increases, each block is shared with nodes. Each round can be checked for sharing by using the path-search algorithm because it has a DAG structure. That is, upon the creation of a new block, a graph search algorithm can be used to detect whether the block is shared on 2n/3 nodes. If we use binary search, we have O(n log n) time complexity because it detects all n nodes that need to be reached. Thus, we determined that the protocol has a total time complexity of $O(n^2 \log n)$ because the blocks that reached 2n/3 through O(n log n) were checked again to reach 2n/3. To determine whether a block reaches an agreement, namely, it has a total time complexity of $O(n^2 \log n)$ to detect the blocks that have been shared.

6.5. Simulation Results

As we mentioned representative protocol *Hashgraph* [17], we compared with our proposed protocol DecisionBFT for various simulation results. We experimented by increasing the total number of nodes to 4, 8, 16 and 32. We also confirmed that the results were significantly different as the number of nodes increased. Figure 22 shows the results of the simulation. Table (c) in Figure 22 is the results of experimentation on a total of three things and a comparison of hashgrpah and DecisionBFT respectively. More detailed panels (a) and (b) in Figure 22 are graphically represented. The panel (a) is the result of experimenting in seconds with the number of nodes in which each node creates an event block. The x-axis is the number of nodes and the y-axis is the generated event blocks per second. It can be seen that the proposed protocol generates blocks of more than hashgraph at the same time. The following are the results for the experiment figure (b). First of all, the x-axis is the result of an increase in the number of nodes, and the y-axis is the corresponding tps time. As the number of nodes increases, hashgraph naturally increases tps and relatively, DecisionBFT can have a fast settlement structure because it uses binary search instead of BFS or DFS. Thus, our proposed protocol has better performance according to an increase the number of nodes. As the number of nodes participating increases geometrically, we expect much better performance.



(a) Event block creation



(b) Transaction per second

# nodes	# event block creation (sec)		tps		latency (sec)	
	Hashgraph	DecisionBFT	Hashgraph	DecisionBFT	Hashgraph	DecisionBFT
4	16.3	46.2	4 080	13,750	0.39	0.32
8	29.6	92	10,960	41,200	0.73	0.67
16	43.9	166.2	29,910	88,200	1.52	0.83
32	71	249.5	55,900	171,500	3.13	1.35

(c) Simulation results according to the number of nodes

Figure 22. Various simulation results.

7. Conclusions

With the aim of realizing distributed ledger technology, we proposed a new family of asynchronous Decision search protocols, namely DecisionBFT. We additionally introduced a consensus structure for a faster consensus. Lamport timestamps were used to improve the intuitiveness and reliability of the topological ordering of event blocks in distributed systems. We added a flag table at each top event block to improve Root detection.

Further, we presented a specific Decision search protocol as an example of DecisionBFT. The DecisionBFT protocol uses a new flag table in each top event block as a shortcut to check for reachability from an event block to a Root along the consensus structure. The path search is used as proof of aBFT consensus. In terms of effectiveness, using the flag table in the DecisionBFT protocol is more effective for consensus compared to path searching approaches. To ensure the distribution of participating nodes, the Decision search protocol defines a new cost function and an algorithm that efficiently and quickly selects peers. We also proposed new algorithms for Root selection and subDecition selection based on the flag table for Decision selection by weight after time consensus ordering.

Based on the DecisionBFT protocol and the new consensus algorithm, our solution can protect against malicious attacks such as forks, double spending, parasite chains, and network control. These protections guarantee the safety of the consensus structure. We can also verify the existence of a Decision within the consensus structure, confirming that the consensus structure guarantees finality and liveliness. The time ordering ensures this guarantee by using the weight value on the flag table. Based on these properties, DSA provides a fair, transparent, and effective consensus algorithm.

We described our formal proof of aBFT for our protocol in Section 4. We also presented formal semantics for the protocol in Section 4.2. Our work is the first to study such concurrent common knowledge semantics [24] in DAG-based protocols. These semantics can be used in systems that require rapid consensus among nodes, such as public or private blockchain environments.

Our framework allows as many as k-nodes to be selected in the gossip environment. This ensures that the selection of additional nodes results in a faster agreement. In our framework, we assigned a value of 3 to k and can improve the current form by adding k from 3 to n.

Future Work

Our generic framework is flexible and extensible and enables the development of a new protocol on the basis thereof. A number of directions for future work are available:

- Using the Decision search protocols, we are designing a fast node synchronization algorithm with k-neighbor broadcasts. The consensus structure and k peer selection make it possible to achieve faster gossip broadcast.
- We are interested in comparing the performance of various gossip strategies, such as random gossip, broadcast gossip, and collection tree protocol for distributed averaging in wireless sensor networks.

A preliminary version of this paper appeared on arXiv [25] and the web [26].

Author Contributions: Conceptualization, S.-M.C., J.P., K.J. and C.P.; Formal analysis, C.P.; Funding acquisition, S.-M.C., J.P., K.J. and C.P.; Investigation, S.-M.C., J.P. and K.J.; Methodology, S.-M.C., J.P.; Supervision, S.-M.C. and J.P.; Writing—original draft, S.-M.C., J.P., K.J. and C.P.; Writing—review & editing, S.-M.C. and C.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF), funded by the Ministry of Education (NRF-2019R1I1A1A01058458).

Acknowledgments: This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2019R1I1A1A01058458). This study was supported by a 2020 Research Grant from Kangwon National University.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Swan, M. *Blockchain: Blueprint for a New Economy*; O'Reilly Media: Sevastopol, CA, USA, 2015.
2. Chen, J.; Micali, S. Algorand. *arXiv* **2016**, arXiv:1607.01341.
3. Gilad, Y.; Hemo, R.; Micali, S.; Vlachos, G.; Zeldovich, N. Algorand: Scaling byzantine agreements for cryptocurrencies. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28–31 October 2017; ACM: New York, NY, USA, 2017; pp. 51–68.
4. Sompolinsky, Y.; Lewenberg, Y.; Zohar, A. SPECTRE: A Fast and Scalable Cryptocurrency Protocol. *IACR Cryptol. ePrint Arch.* **2016**, *2016*, 1159.
5. Sompolinsky, Y.; Zohar, A. PHANTOM, GHOSTDAG: Two Scalable BlockDAG Protocols. 2020. Available online: https://eprint.iacr.org/2018/104.pdf (accessed on 19 August 2020).
6. Lamport, L.; Shostak, R.; Pease, M. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* **1982**, *4*, 382–401. [CrossRef]
7. Aspnes, J. Randomized protocols for asynchronous consensus. *Distrib. Comput.* **2003**, *16*, 165–175. [CrossRef]
8. Lamport, L. Paxos made simple. *ACM Sigact News* **2001**, *32*, 18–25.
9. Fischer, M.J.; Lynch, N.A.; Paterson, M. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* **1985**, *32*, 374–382. [CrossRef]
10. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. Available online: https://bitcoin.org/bitcoin.pdf (accessed on 19 August 2020).
11. Sunny King, S.N. PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake. 2012. Available online: https://decred.org/research/king2012.pdf (accessed on 19 August 2020).
12. Lerner, S.D. DagCoin. 2015. Available online: https://bitslog.files.wordpress.com/2015/09/dagcoin-v41.pdf (accessed on 19 August 2020).
13. Chevalier, P.; Kaminski, B.; Hutchison, F.; Ma, Q.; Sharma, S. Protocol for Asynchronous, Reliable, Secure and Efficient Consensus (PARSEC). *arXiv* **2018**, arXiv:1907.11445.
14. Li, C.; Li, P.; Zhou, D.; Xu, W.; Long, F.; Yao, A. Scaling Nakamoto Consensus to Thousands of Transactions per Second. *arXiv* **2018**, arXiv:1805.03870.
15. Popov, S.; Saa, O.; Finardi, P. Equilibria in the Tangle. *arXiv* **2019**, arXiv:1712.05385.
16. Churyumov, A. Byteball: A Decentralized System for Storage and Transfer of Value. 2016. Available online: https://obyte.org/Byteball.pdf (accessed on 19 August 2020).
17. Baird, L. Hashgraph Consensus: Fair, Fast, Byzantine Fault Tolerance. 2016. Available online: https://www.swirlds.com/wp-content/uploads/2016/06/2016-05-31-Swirlds-Consensus-Algorithm-TR-2016-01.pdf (accessed on 19 August 2020).
18. Castro, M.; Liskov, B. Practical Byzantine Fault Tolerance. In Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99), New Orleans, LA, USA, 22–25 February 1999; USENIX Association: Berkeley, CA, USA, 1999; pp. 173–186.
19. Kotla, R.; Alvisi, L.; Dahlin, M.; Clement, A.; Wong, E. Zyzzyva: Speculative byzantine fault tolerance. *ACM SIGOPS Oper. Syst. Rev.* **2007**, *41*, 45–58. [CrossRef]
20. Miller, A.; Xia, Y.; Croman, K.; Shi, E.; Song, D. The honey badger of BFT protocols. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; ACM: New York, NY, USA, 2016; pp. 31–42.
21. Danezis, G.; Hrycyszyn, D. Blockmania: From Block DAGs to Consensus. *arXiv* **2018**, arXiv:1809.01620.
22. LeMahieu, C. Nano: A Feeless Distributed Cryptocurrency Network. 2017. Available online: https://content.nano.org/whitepaper/Nano_Whitepaper_en.pdf (accessed on 19 August 2020).
23. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* **1978**, *21*, 558–565.
24. Panangaden, P.; Taylor, K. Concurrent common knowledge: Defining agreement for asynchronous systems. *Distrib. Comput.* **1992**, *6*, 73–93. [CrossRef]

25. Choi, S.M.; Park, J.; Nguyen, Q.; Cronje, A. Fantom: A scalable framework for asynchronous distributed systems. *arXiv* **2018**, arXiv:1810.10360.
26. Fantom-Foundation/Fantom-Documentation. 2018. Available online: https://github.com/Fantom-foundation/fantom-documentation/wiki (accessed on 19 August 2020).