FANTOM: A SCALABLE FRAMEWORK FOR ASYNCHRONOUS DISTRIBUTED SYSTEMS

A PREPRINT

Sang-Min Choi, Jiho Park, Quan Nguyen, and Andre Cronje

FANTOM Lab
FANTOM Foundation

February 27, 2019

ABSTRACT

We describe *Fantom*, a framework for asynchronous distributed systems. *Fantom* is based on the Lachesis Protocol [1], which uses asynchronous event transmission for practical Byzantine fault tolerance (pBFT) to create a leaderless, scalable, asynchronous Directed Acyclic Graph (DAG).

We further optimize the *Lachesis Protocol* by introducing a permission-less network for dynamic participation. Root selection cost is further optimized by the introduction of an n-row flag table, as well as optimizing path selection by introducing domination relationships.

We propose an alternative framework for distributed ledgers, based on asynchronous partially ordered sets with logical time ordering instead of blockchains.

This paper builds upon the original proposed family of *Lachesis-class* consensus protocols. We formalize our proofs into a model that can be applied to abstract asynchronous distributed system.

Keywords Consensus algorithm · Byzantine fault tolerance · Lachesis protocol · OPERA chain · Lamport timestamp · Main chain · Root · Clotho · Atropos · Distributed Ledger · Blockchain

Contents

1 Introduction

Blockchain has emerged as a technology for secure decentralized transaction ledgers with broad applications in financial systems, supply chains and health care. *Byzantine* fault tolerance [2] is addressed in distributed database systems, in which up to one-third of the participant nodes may be compromised. Consensus algorithms [3] ensures the integrity of transactions between participants over a distributed network [2] and is equivalent to the proof of *Byzantine* fault tolerance in distributed database systems [4, 5].

A large number of consensus algorithms have been proposed. For example; the original Nakamoto consensus protocol in Bitcoin uses Proof of Work (PoW) [6]. Proof Of Stake (PoS) [7, 8] uses participants' stakes to generate the blocks respectively. Our previous paper gives a survey of previous DAG-based approaches [1].

In the previous paper [1], we introduced a new consensus protocol, called L_0. The protocol L_0 is a DAG-based asynchronous non-deterministic protocol that guarantees pBFT. L_0 generates each block asynchronously and uses the OPERA chain (DAG) for faster consensus by confirming how many nodes share the blocks.

The Lachesis protocol as previously proposed is a set of protocols that create a directed acyclic graph for distributed systems. Each node can receive transactions and batch them into an event block. An event block is then shared with it's peers. When peers communicate they share this information again and thus spread this information through the network. In BFT systems we would use a broadcast voting approach and ask each node to vote on the validity of each block. This event is synchronous in nature. Instead we proposed an asynchronous system where we leverage the concepts of distributed common knowledge, dominator relations in graph theory and broadcast based gossip to achieve a local view with high probability of being a global view. It accomplishes this asynchronously, meaning that we can increase throughput near linearly as nodes enter the network.

In this work, we propose a further enhancement on these concepts and we formalize them so that they can be applied to any asynchronous distributed system.

1.1 Contributions

In summary, this paper makes the following contributions:

- We introduce the n-row flag table for faster root selection of the Lachesis Protocol.
- We define continuous consistent cuts of a local view to achieve consensus.
- We present proof of how domination relationships can be used for share information.
- We formalize our proofs that can be applied to any generic asynchronous DAG solution.

1.2 Paper structure

The rest of this paper is organised as follows. Section 2 describes our Fantom framework. Section 3 presents the protocol implementation. Section 4 concludes. Proof of Byzantine fault tolerance is described in Section 5.2.

2 Preliminaries

The protocol is run via nodes representing users' machines which together create a network. The basic units of the protocol are called event blocks - a data structure created by a single node to share transaction and user information with the rest of the network. These event blocks reference previous event blocks that are known to the node. This flow or stream of information creates a sequence of history.

The history of the protocol can be represented by a directed acyclic graph $G = (V, E)$, 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 left-to-right of the graph, so left vertices represent earlier events in history.

For a graph G, a path p in G is a sequence of vertices $(v_1, v_2, \ldots, v_k)$ 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.

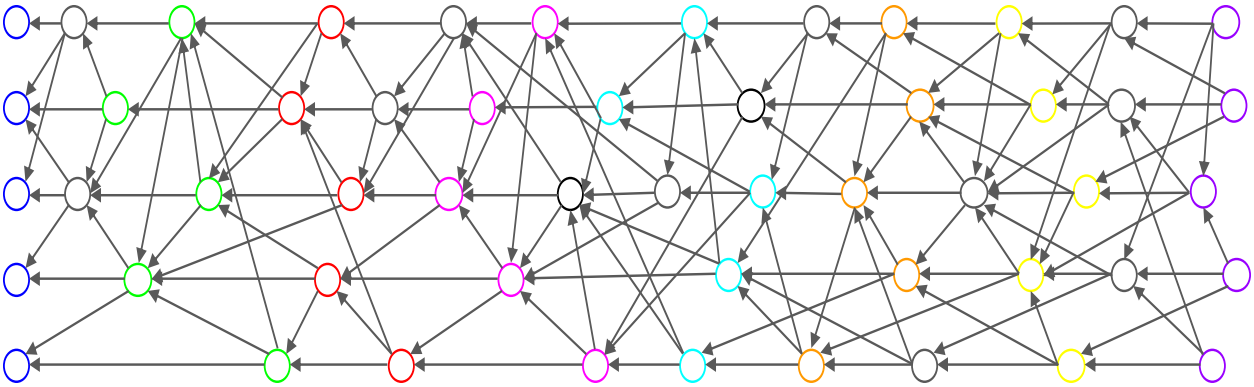

Figure 1: An Example of OPERA Chain

Figure 1 shows an example of an OPERA chain (DAG) constructed through the Lachesis protocol. Event blocks are representated by circles. Blocks of the same frame have the same color.

2.1 Basic Definitions

(Lachesis) The set of *protocols*

(Node) Each machine that participates in the Lachesis protocol is called a *node*. Let n denote the total number of nodes.

(k) A *constant* defined in the system.

(Peer node) A node n_i has k *peer nodes*.

(Process) A 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.

(Channel) A 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.

(Event block) Each node can create event blocks, send (receive) messages to (from) other nodes.The structure of an event block includes the signature, generation time, transaction history, and hash information to references.

All nodes can create event blocks. The information of the referenced event blocks can be copied by each node. The first event block of each node is called a *leaf event*.

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 the references is self-reference, and the other k-1 references point to the top events of n_i's k-1 peer nodes.

(Top event) An event v is a top event of a node n_i if there is no other event in n_i referencing v.

(Height Vector) The height vector is the number of event blocks *created* by the i-th node.

(In-degree Vector) The in-degree vector refers to the number of *edges* from other event blocks created by other nodes to the top event block of this node. The top event block indicates the most recently created event block by this node.

(Ref) An event v_r is called "ref" of event v_c if the reference hash of v_c points to the event v_r. Denoted by $v_c \hookrightarrow^r v_r$. For simplicity, we can use $\hookrightarrow$ to denote a reference relationship (either $\hookrightarrow^r$ or $\hookrightarrow^s$).

(Self-ref) An event v_s is called "self-ref" of event v_c, if the self-ref hash of v_c points to the event v_s. Denoted by $v_c \hookrightarrow^s v_s$.

(k references) Each event block has at least k references. One of the references is self-reference, and the other k-1 references point to the top events of n_i's k-1 peer nodes.

(Self-ancestor) An event block v_a is self-ancestor of an event block v_c if there is a sequence of events such that $v_c \hookrightarrow^s v_1 \hookrightarrow^s \ldots \hookrightarrow^s v_m \hookrightarrow^s v_a$. Denoted by $v_c \hookrightarrow^{sa} v_a$.

(Ancestor) An event block v_a is an ancestor of an event block v_c if there is a sequence of events such that $v_c \hookrightarrow v_1 \hookrightarrow \ldots \hookrightarrow v_m \hookrightarrow v_a$. Denoted by $v_c \hookrightarrow^a v_a$.

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

(Flag Table) The flag table is a $n \times k$ matrix, where n is the number of nodes and k is the number of roots that an event block can reach. If an event block e created by i-th node can reach j-th root, then the flag table stores the hash value of the j-th root.

2.2 Lamport timestamps

Our Lachesis protocols relies on Lamport timestamps to define a topological ordering of event blocks in OPERA chain. By using Lamport timestamps, we do not rely on physical clocks to determine a 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 called 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 comes before 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, a 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 that the Clock Condition implies that if $v \rightarrow v'$ then $v \Rightarrow v'$.

We use this total ordering in our Lachesis protocol. This ordering is used to determine consensus time, as described in Section 3.

(Happened-Immediate-Before) An event block v_x is said Happened-Immediate-Before an event block v_y if v_x is a (self-) ref of v_y. Denoted by $v_x \mapsto v_y$.

(Happened-before) An event block v_x is said Happened-Before an event block v_y if v_x is a (self-) ancestor of v_y. Denoted by $v_x \rightarrow v_y$.

Happened-before is the relationship between nodes which have event blocks. If there is a path from an event block v_x to v_y, then v_x Happened-before v_y. "v_x Happened-before v_y" means that the node creating v_y knows event block v_x. This relation is the transitive closure of happens-immediately-before. Thus, an event v_x happened before an event v_y if one of the followings happens: (a) $v_y \hookrightarrow^s v_x$, (b) $v_y \hookrightarrow^r v_x$, or (c) $v_y \hookrightarrow^a v_x$. The happened-before relation of events form an acyclic directed graph $G' = (V, E')$ such that an edge $(v_i, v_j) \in E'$ has a reverse direction of the same edge in E.

(Concurrent) Two event blocks v_x and v_y are said concurrent if neither of them happened before the other. Denoted by $v_x \parallel v_y$.

Given two vertices v_x and v_y both contained in two OPERA chains (DAGs) G_1 and G_2 on two nodes. We have the following:

(1) $v_x \rightarrow v_y$ in G_1 if $v_x \rightarrow v_y$ in G_2.

(2) $v_x \parallel v_y$ in G_1 if $v_x \parallel v_y$ in G_2.

2.3 State Definitions

Each node has a local state, a collection of histories, messages, event blocks, and peer information, we describe the components of each.

(State) A state of a process i is denoted by s^i_j.

(Local State) 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 if the reference blocks exist before it. From a local state s^i_j, one can reconstruct a unique DAG. That is, the mapping from a local state s^i_j into a DAG is *injective* or one-to-one. Thus, for Fantom, we can simply denote the j-th local state of a process i by the DAG g^i_j (often we simply use G_i to denote the current local state of a process i).

(Action) An action is a function from one local state to another local state.

Generally speaking, an action can be one of: a $send(m)$ action where m is a message, a $receive(m)$ action, and 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 Lachesis protocol, B consists of the content of an event block v.

Semantics-wise, in Lachesis, there are two actions that can change a process's local state: creating a new event and receiving an event from another process.

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

(Local history) A local history h_i of process i is a (possibly infinite) sequence of alternating local states — beginning with a distinguished initial state. A set H_i of 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 Lachesis protocol, we use append-only semantics. The local history may be equivalently described as either of the following:

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

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

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

In Lachesis, a local history is equivalently expressed as:

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

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

(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 talk about 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.

2.4 Consistent Cut

Consistent cuts represent the concept of scalar time in distributed computation, it is possible to distinguish between a "before" and an "after".

In the Lachesis protocol, an OPERA chain $G = (V, E)$ is a directed acyclic graph (DAG). V is a set of vertices and E is a set of edges. DAG is a directed graph with no cycle. There is no path that has source and destination at the same vertex. 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 is the set of possible local histories for each process i; a set A of asynchronous runs; a set M of all messages.

Each process / node in Lachesis selects k other nodes as peers. For certain gossip protocol, nodes may be constrained to gossip with its k peers. In such a case, the set of channels C can be modelled 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 DAG-based protocol in general or in Lachesis protocol in particular, in the same manner as in the CCK paper [9].

(Consistent cut) A consistent cut of a run σ is any global state such that if $v^i_x \rightarrow v^j_y$ and v^j_y is in the global state, then v^i_x is also in the global state. Denoted by $\mathbf{c}(\sigma)$.

The concept of consistent cut formalizes such a global state of a run. A consistent cut consists of all consistent DAG chains. A received event block exists in the global state implies the existence of the original event block. Note that a consistent cut is simply a vector of local states; we will use the notation $\mathbf{c}(\sigma)[i]$ to indicate the local state of i in cut $\mathbf{c}$ of run σ.

A message chain of an asynchronous run is a sequence of messages $m_1, m_2, m_3, \dots$, such that, for all i, $receive(m_i) \rightarrow send(m_{i+1})$. Consequently, $send(m_1) \rightarrow receive(m_1) \rightarrow send(m_2) \rightarrow receive(m_2) \rightarrow send(m_3) \dots$.

The formal semantics of an asynchronous system is 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 $\mathbf{c}(\sigma) \vdash p$.

(Equivalent cuts) Two cuts $\mathbf{c}(\sigma)$ and $\mathbf{c}'(\sigma')$ are equivalent with respect to i if:

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

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

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

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

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

(Majority concurrently knows) The next modal operator is written M^C and stands for "majority concurrently knows." The definition of $M^C(\phi)$ is as follows.

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

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

This is adapted from the "everyone concurrently knows" in CCK paper [9]. In the presence of one-third of faulty nodes, the original operator "everyone concurrently knows" is sometimes not feasible. Our modal operator $M^C(\phi)$ fits precisely the semantics for BFT systems, in which unreliable processes may exist.

(Concurrent common knowledge) The last modal operator is concurrent common knowledge (CCK), denoted by C^C. $C^C(\phi)$ 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 that same state of knowledge, with respect to ϕ, along some cut of the run. In other words, we want a state of knowledge X satisfying: $X = M^C(\phi \wedge X)$. C^C will be defined semantically as the weakest such fixed point, namely as 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 some cut in the same asynchronous run σ including i's local state, such that ϕ is true in that cut.

Note that ϕ implies $P_i(\phi)$. But it 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 some cut $\mathbf{c}(\sigma)$. A process cannot distinguish which cut, of the perhaps many cuts that are in the run and consistent with its local state, satisfies ϕ; it can only know the existence of such a cut.

(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.

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

Fact ϕ is valid, denoted $\vdash \phi$, if ϕ is valid in all systems, i.e. $(\forall \Sigma)(\Sigma \vdash \phi)$.

(Local fact) A fact ϕ is local to process i in system Σ if $\Sigma \vdash (\phi \Rightarrow K_i \phi)$.

2.5 Dominator (graph theory)

In a graph $G = (V, E, r)$ a dominator is the relation between two vertices. A vertex v is dominated by another vertex w, if every path in the graph from the root r to v have to go through w. Furthermore, the immediate dominator for a vertex v is the last of v's dominators, which every path in the graph have to go through to reach v.

(Pseudo top) A pseudo vertex, called top, is the parent of all top event blocks. Denoted by $\top$.

(Pseudo bottom) A pseudo vertex, called bottom, is the child of all leaf event blocks. Denoted by $\bot$.

With the pseudo vertices, we have $\bot$ happened-before all event blocks. Also all event blocks happened-before $\top$. That is, for all event v_i, $\bot \rightarrow v_i$ and $v_i \rightarrow \top$.

(Dom) An event v_d dominates an event v_x if every path from $\top$ to v_x must go through v_d. Denoted by $v_d \gg v_x$.

(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. Denoted by $v_d \gg^s v_x$.

(Domfront) A vertex v_d is said "domfront" a vertex v_x if v_d dominates an immediate predecessor of v_x, but v_d does not strictly dominate v_x. Denoted by $v_d \gg^f v_x$.

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

2.6 OPERA chain (DAG)

The core idea of the Lachesis protocol is to use a DAG-based structure, called the OPERA chain for our consensus algorithm. In the Lachesis 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. The communication between nodes is asynchronous.

Let n be the number of participant nodes. For consensus, the algorithm examines whether an event block is *dominated* by $2n/3$ nodes, where n is the number of all nodes. The Happen-before relation of event blocks with $2n/3$ nodes means that more than two-thirds of all nodes in the OPERA chain know the event block.

The OPERA chain (DAG) is the local view of the DAG held by each node, this local view is used to identify topological ordering, select Clotho, and create time consensus through Atropos selection. OPERA chain 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, $v_i \hookrightarrow v_j$.

(Leaf) The first created event block of a node is called a leaf event block.

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

(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).

(Frame) Frame f_i is a natural number that separates Root sets. The root set at frame f_i is denoted by R_i.

(Consistent chains) OPERA chains G_1 and G_2 are consistent if for any event v contained in both chains, $G_1[v] = G_2[v]$. Denoted by $G_1 \sim G_2$.

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

If two nodes have OPERA chains 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, so both OPERA chains 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 OPERA chains are consistent.

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

(Consistent chain) A global consistent chain G^C is a chain if $G^C \sim G_i$ for all G_i.

We denote $G \sqsubseteq G'$ to stand for G is a subgraph of G'. Some properties of G^C are given as follows:

(1) $\forall G_i\ (G^C \sqsubseteq G_i)$.

(2) $\forall v \in G^C\ \forall G_i\ (G^C[v] \sqsubseteq G_i[v])$.

(3) $(\forall v_c \in G^C)\ (\forall v_p \in G_i)\ ((v_p \rightarrow v_c) \Rightarrow v_p \in G^C)$.

(Consistent root) Two chains G_1 and G_2 are root consistent, if for every v contained in both chains, v is a root of j-th frame in G_1, then v is a root of j-th frame in G_2.

By consistent chains, if $G_1 \sim G_2$ and v belongs to both chains, then $G_1[v] = G_2[v]$. We can prove the 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 $\forall u \in S\ (u \in G_2)$. Since 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 happens before v in G_2. So v belongs to f_{k+1} in G_2.

Thus, all nodes have the same consistent root sets, which are the root sets in G^C. Frame numbers are consistent for all nodes.

(Flag table) A flag table stores reachability from an event block to another root. The sum of all reachabilities, namely all values in flag table, indicates the number of reachabilities from an event block to other roots.

(Consistent flag table) For any top event v in both OPERA chains G_1 and G_2, and $G_1 \sim G_2$, then the flag tables of v are consistent if they are the same in both chains.

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

(Clotho) A root r_k in the frame f_{a+3} can nominate a root r_a as Clotho if more than 2n/3 roots in the frame f_{a+1} dominate r_a and r_k dominates the roots in the frame f_{a+1}.

Each node nominates a root into Clotho via the flag table. If all nodes have an OPERA chain with same shape, the values in flag table will be equal to each other in OPERA chain. Thus, all nodes nominate the same root into Clotho since the OPERA chain of all nodes has same shape.

(Atropos) An Atropos is assigned consensus time through the Lachesis consensus algorithm and is utilized for determining the order between event blocks. Atropos blocks form a Main-chain, which allows time consensus ordering and responses to attacks.

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} selects 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 Reselection process. Based on the Reselection process, the time consensus algorithm can reach agreement. However, there is a possibility that consensus time candidate does not reach agreement [10]. To solve this problem, time consensus algorithm includes minimal selection frame per next h frame. In minimal value selection algorithm, each root selects minimum value among values collected from previous root set. Thus, the consensus time reaches consensus by time consensus algorithm.

(Main-chain (Blockchain)) For faster consensus, *Main-chain* is a special sub-graph of the OPERA chain (DAG).

The Main chain — a core subgraph of OPERA chain, plays the important role of ordering the event blocks. The Main chain stores shortcuts to connect between the Atropos. After the topological ordering is computed over all event blocks through the Lachesis protocol, Atropos blocks are determined and form the Main chain. To improve path searching, we use a flag table — a local hash table structure as a cache that is used to quickly determine the closest root to a event block.

In the OPERA chain, an event block is called a *root* if the event block is linked to more than two-thirds of previous roots. A leaf vertex is also a root itself. With root event blocks, we can keep track of "vital" blocks that $2n/3$ of the network agree on.

Figure 2 shows an example of the Main chain composed of Atropos event blocks. In particular, the Main chain consists of Atropos blocks that are derived from root blocks and so are agreed by $2n/3$ of the network nodes. Thus, this guarantees that at least $2n/3$ of nodes have come to consensus on this Main chain.

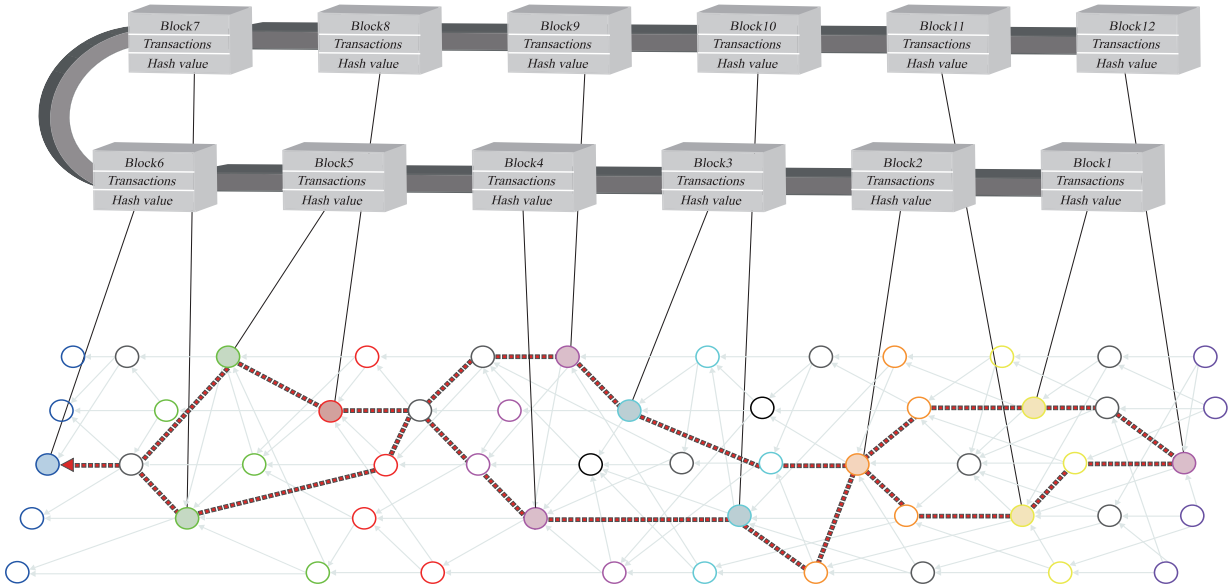

Figure 2: An Example of Main-chain

Each participant node has a copy of the Main chain and can search consensus position of its own event blocks. Each event block can compute its own consensus position by checking the nearest Atropos event block. Assigning and searching consensus position are introduced in the consensus time selection section.

The Main chain provides quick access to the previous transaction history to efficiently process new incoming event blocks. From the Main chain, information about unknown participants or attackers can be easily viewed. The Main chain can be used efficiently in transaction information management by providing quick access to new event blocks that have been agreed on by the majority of nodes. In short, the Main-chain gives the following advantages:

- All event blocks or nodes do not need to store all information. It is efficient for data management.

- Access to previous information is efficient and fast.

Based on these advantages, OPERA chain can respond strongly to efficient transaction treatment and attacks through its Main-chain.

3 Lachesis Protocol

Algorithm 1 Main Procedure

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

Algorithm 1 shows the pseudo algorithm for the Lachesis core procedure. The algorithm consists of two parts and runs them in parallel.

- In part one, each node requests synchronization and creates event blocks. 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 line 4 and 5, the node synchronizes the OPERA chain (DAG) with the other nodes. Line 6 runs the Event block creation, at which step the node creates an event block and checks whether it is a root. The node then broadcasts the created event block to all other known nodes in line 7. The step in this line is optional. In line 8 and 9, Clotho selection and Atropos time consensus algorithms are invoked. The algorithms determines whether the specified root can be a Clotho, assign the consensus time, and then confirm the Atropos.

- The second part is to respond to synchronization requests. In line 10 and 11, the node receives a synchronization request and then sends its response about the OPERA chain.

3.1 Peer selection algorithm

In order to create an event block, a node needs to select k other nodes. Lachesis protocols does not depend on how peer nodes are selected. One simple approach can use a random selection from the pool of n nodes. The other approach is to define some criteria or cost function to select other peers of a node.

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

3.2 Dynamic participants

Our Lachesis protocol allows an arbitrary number of participants to dynamically join the system. The OPERA chain (DAG) can still operate with new participants. Computation on flag tables is set based and independent of which and how many participants have joined the system. Algorithms for selection of Roots, Clothos and Atroposes are flexible enough and not dependence on a fixed number of participants.

3.3 Peer synchronization

We describe an algorithm that synchronizes events between the nodes.

Algorithm 2 EventSync

1: **procedure** SYNC-EVENTS()
2: Node n_1 selects random peer to synchronize with
3: n_1 gets local known events (map[int]int)
4: n_1 sends RPC request Sync request to peer
5: n_2 receives RPC requestSync request
6: n_2 does an EventDiff check on the known map (map[int]int)
7: n_2 returns unknown events, and map[int]int of known events to n_1

The algorithm assumes that a node always needs the events in topological ordering (specifically in reference to the lamport timestamps), an alternative would be to use an inverse bloom lookup table (IBLT) for completely potential randomized events.

Alternatively, one can simply use a fixed incrementing index to keep track of the top event for each node.

3.4 Node Structure

This section gives an overview of the node structure in Lachesis.

Each node has a height vector, in-degree vector, flag table, frames, clotho check list, max-min value, main-chain (blockchain), and their own local view of the OPERA chain (DAG). The height vector is the number of event blocks created by the i-th node. The in-degree vector refers to the number of edges from other event blocks created by other nodes to the top event block of this node. The top event block indicates the most recently created event block by this node. The flag table is a nxk matrix, where n is the number of nodes and k is the number of roots that an event block can reach. If an event block e created by i-th node can reach j-th root, then the flag table stores the hash value of the j-th root. Each node maintains the flag table of each top event block.

Frames store the root set in each frame. Clotho check list has two types of check points; Clotho candidate (CC) and Clotho (C). If a root in a frame is a CC, a node check the CC part and if a root becomes Clotho, a node check C part. Max-min value is timestamp that addresses for Atropos selection. The Main-chain is a data structure storing hash values of the Atropos blocks.

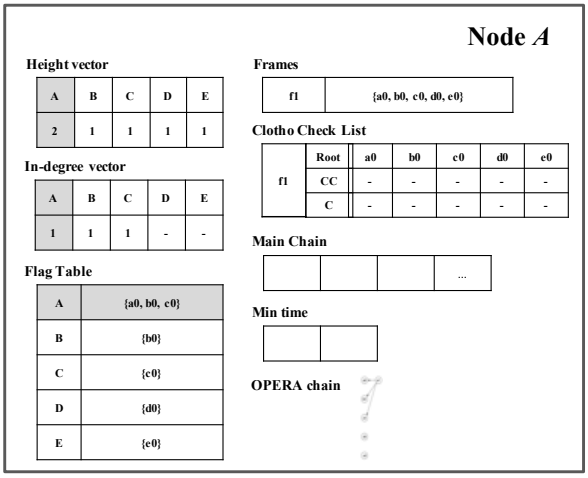

Figure 3: An Example of Node Structure

Figure 3 shows an example of the node structure component of a node A. In the figure, each value excluding self height in the height vector is 1 since the initial state is shared to all 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. The in-degrees of node A, B, and C are 1. In flag table, node A knows other two root hashes since the top event block can reach those two roots. Node A also knows that other nodes know their own roots. In the example situation there is no clotho candidate and Clotho, and thus clotho check list is empty. The main-chain and max-min value are empty for the same reason as clotho check list.

3.5 Peer selection algorithm via Cost function

We define three versions of the Cost Function (C_F). Version one is focused around updated information share and is discussed below. The other two versions are focused on root creation and consensus facilitation, these will be discussed in a following paper.

We define a Cost Function (C_F) for preventing the creation of lazy nodes. A lazy node is a node that has a lower work portion in the OPERA chain (has created fewer event blocks). When a node creates an event block, the node selects other nodes with low value outputs from the cost function and refers to the top event blocks of the reference nodes. An equation (1) of C_F is as follows,

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

where I and H denote values of in-degree vector and height vector respectively. If the number of nodes with the lowest C_F is more than k, one of the nodes is selected at random. The reason for selecting high H is that we can expect a high possibility to create a root because the high H indicates that the communication frequency of the node had more opportunities than others with low H. Otherwise, the nodes that have high C_F (the case of $I > H$) have generated fewer event blocks than the nodes that have low C_F..

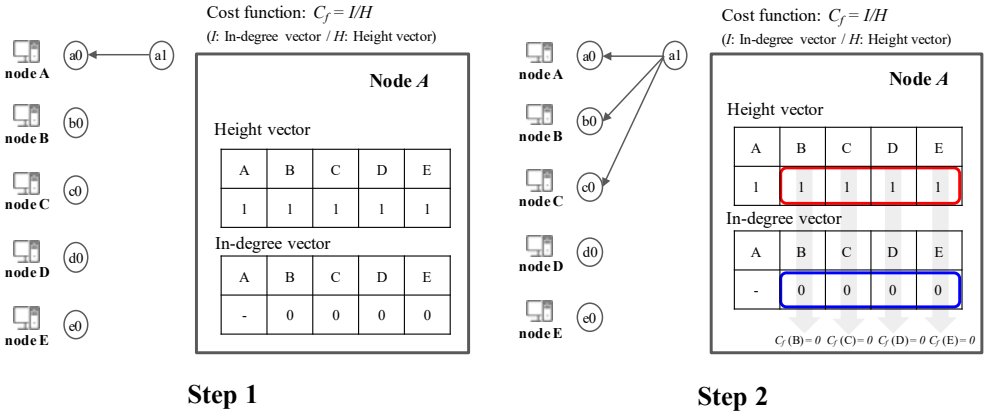

Figure 4: An Example of Cost Function 1

Figure 4 shows an example of the node selection based on the cost function after the creation of leaf events by all nodes. In this example, there are 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 4 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 are 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.

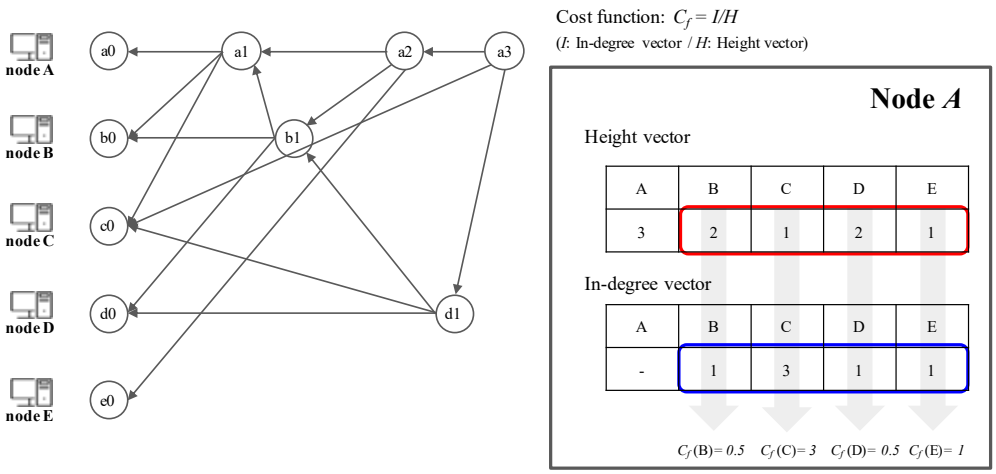

Figure 5: An Example of Cost Function 2

Figure 5 shows an example of the node selection after a few steps of the simulation in Figure 4. In Figure 5, the 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 other nodes have the same values. Because of this, node A first selects node B and randomly selects other nodes among nodes C, D, and E.

The height of node D in the example is 2 (leaf event and event block v_4). On the other hand, the height of node D in node structure of A is 1. Node A is still not aware of the presence of the event block v_4. It means that there is no path from the event blocks created by node A to the event block v_4. Thus, node A has 1 as the height of node D.

Algorithm 3 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 $\leftarrow INF$
5: $s_{ref} \leftarrow$ 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 $\leftarrow c_f$
10: $s_{ref} \leftarrow$ k
11: **else if** min_cost $equal$ c_f **then**
12: $s_{ref} \leftarrow s_{ref} \cup k$
13: $ref \leftarrow$ random select in s_{ref}

Algorithm 3 shows the selecting algorithm for selecting reference nodes. The algorithm operates for each node to select a communication partner from other nodes. Line 4 and 5 set min_cost and S_{ref} to initial state. Line 7 calculates the cost function c_f for each node. In line 8, 9, and 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. Line 11 and 12 append the ID of each node to S_{ref} if min_cost equals c_f. Finally, line 13 selects randomly 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 of all event blocks known by them. A node creates an event block by referring to the top event block of the reference node. The Lachesis protocol works and communicates asynchronously. This allows a node to create an event block asynchronously even when another node creates an event block. The communication between nodes does not allow simultaneous communication with the same node.

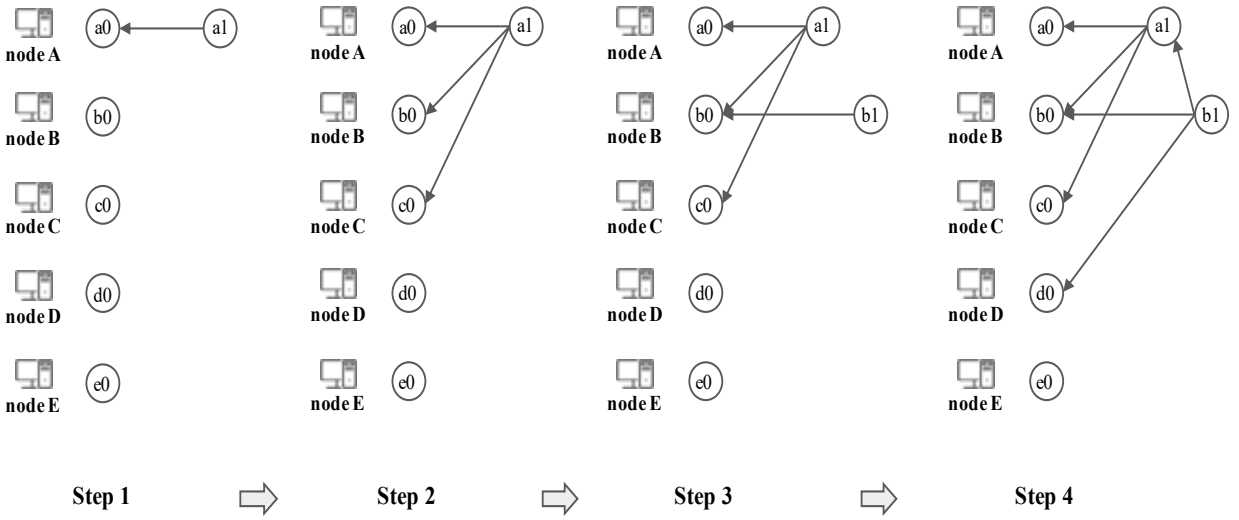

Figure 6: An Example of Node Selection

Figure 6 shows an example of the node selection in Lachesis protocol. In this example, there are five nodes (A, B, C, D, and E) and each node generates the first event blocks, called 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 called leaf event, thus the height vector and the in-degree vector in node A has same values. In other words, the heights of each node are 1 and in-degrees are 0. Node A randomly select the other two nodes and connects a_1 to the top two event blocks from the other two nodes. Step 2 shows the situation after connections. In this example, node A select node 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 of the step 2.

After that, in the example, node B generates a new event block b_1 and also calculates the cost function. B randomly select the other two nodes; A, and D, since B only has information of the leaf events. Node B requests to A and D to connect b_1, then nodes A and D send information for their top event blocks to node B as 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.

3.6 Event block creation

In the Lachesis protocol, every node can create an event block. Each event block refers to other k event blocks using their hash values. In the 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 blocks of its own node.
2. One reference should be made to a self-ref that references 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.

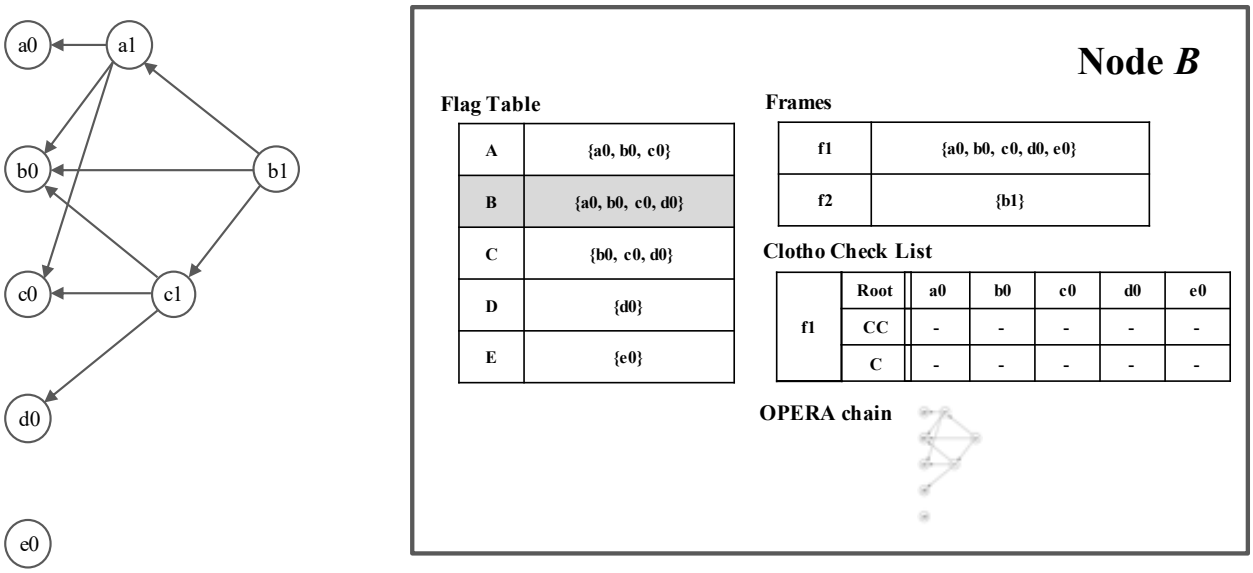

Figure 7: An Example of Event Block Creation with Flag Table

Figure 7 shows the example of an event block creation with a flag table. In this example the recent created event block is b_1 by node B. The figure shows the node structure of node B. We omit the other information such as height and in-degree vectors since we only focus on the change of the flag table with the event block creation in this example. The flag table of b_1 in Figure 7 is updated with the information of the previous connected event blocks a_1, b_0, and c_1. Thus, the set of the flag table is the results of 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).

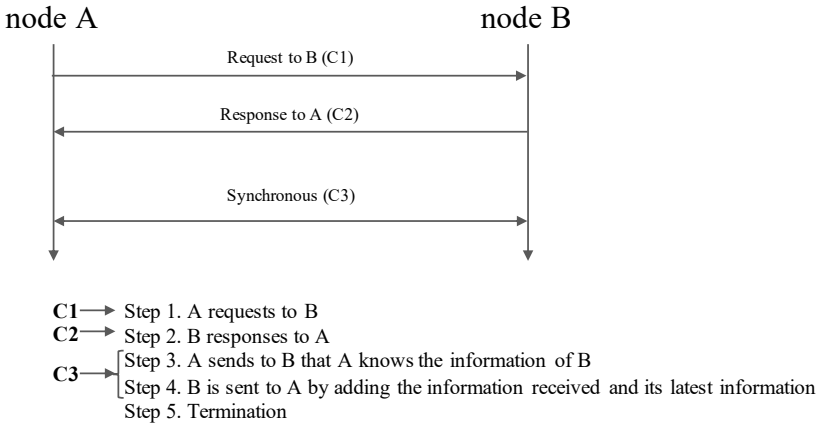

Figure 8: An Example of Communication Process

Figure 8, shows the communication process is divided into five steps for two nodes to create an event block. Simply, a node A requests to B. then, B responds to A directly.

3.7 Topological ordering of events using Lamport timestamps

Every node has a physical clock and it needs physical time to create an event block. However, for consensus, Lachesis protocols relies on a logical clock for each node. For the purpose, we use *"Lamport timestamps"* [11] to determine the time ordering between event blocks in a asynchronous distributed system.

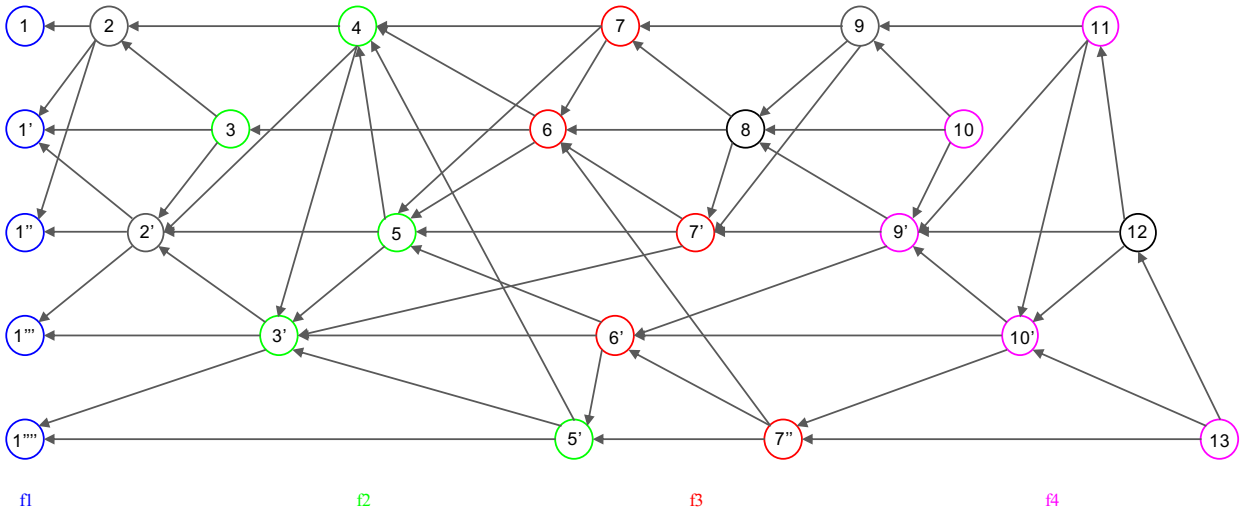

Figure 9: An example of Lamport timestamps

The Lamport timestamps algorithm is as follows:

1. Each node increments its count value before creating an event block.
2. When sending a message include its count value, receiver should consider which sender's message is received and increments its count value.
3. If 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 current counter is greater than the received count value from another node, then the current count value is updated.

We use the Lamport's algorithm to enforce a topological ordering of event blocks and use it in the Atropos selection algorithm.

Since an event block is created based on logical time, the sequence between each event blocks is immediately determined. Because the Lamport timestamps algorithm gives a partial order of all events, the whole time ordering process can be used for Byzantine fault tolerance.

3.8 Domination Relation

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

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:

A vertex v_d belongs to a $\frac{2}{3}$-dom set within the 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}.

The $\frac{2}{3}$-dom set D_i is the same with the root set R_i, for all nodes.

3.9 Examples of domination relation in DAGs

This section gives several examples of DAGs and the domination relation between their event blocks.

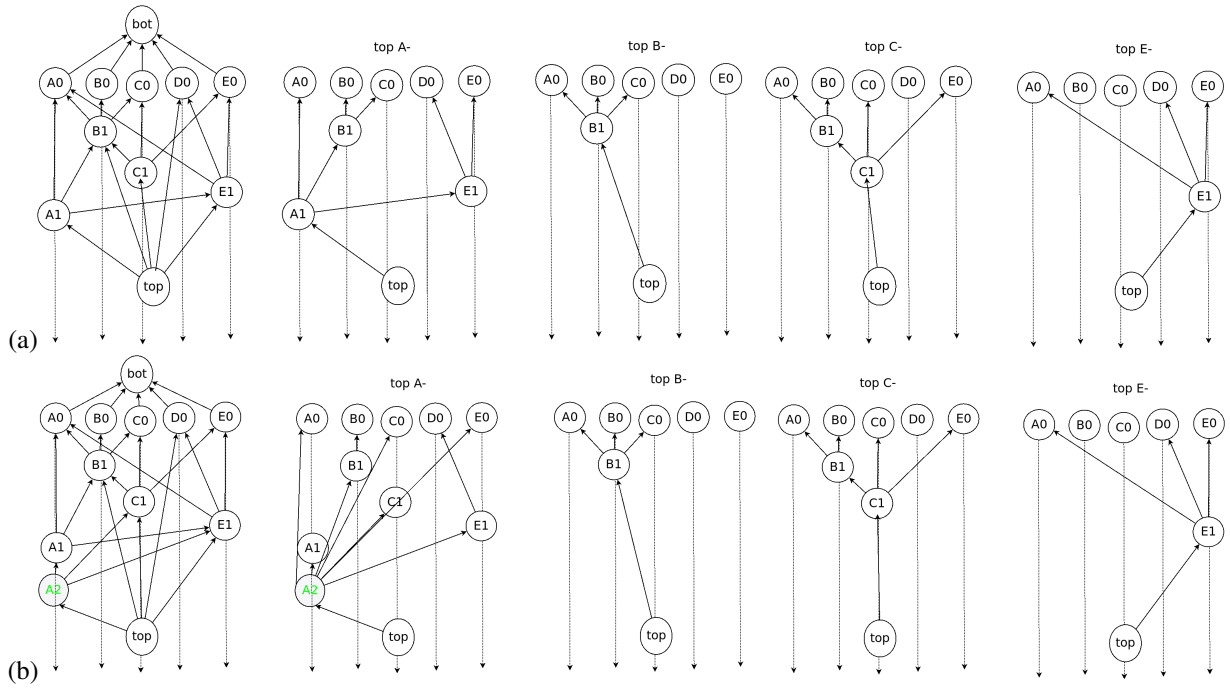

Figure 10: Examples of OPERA chain and dominator tree

Figure 10 shows an examples of a DAG and dominator trees.

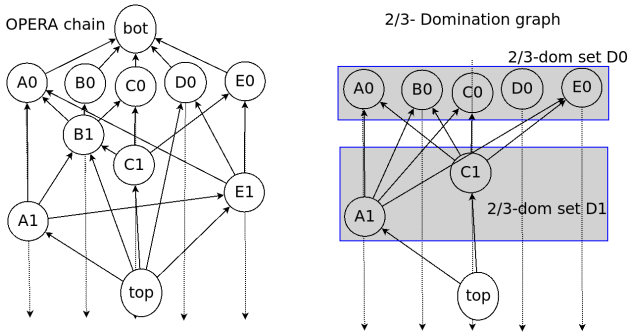

Figure 11: An example of OPERA chain and its 2/3 domination graph. The $\frac{2}{3}$-dom sets are shown in grey.

Figure 11 depicts an example of a DAG and 2/3 dom sets.

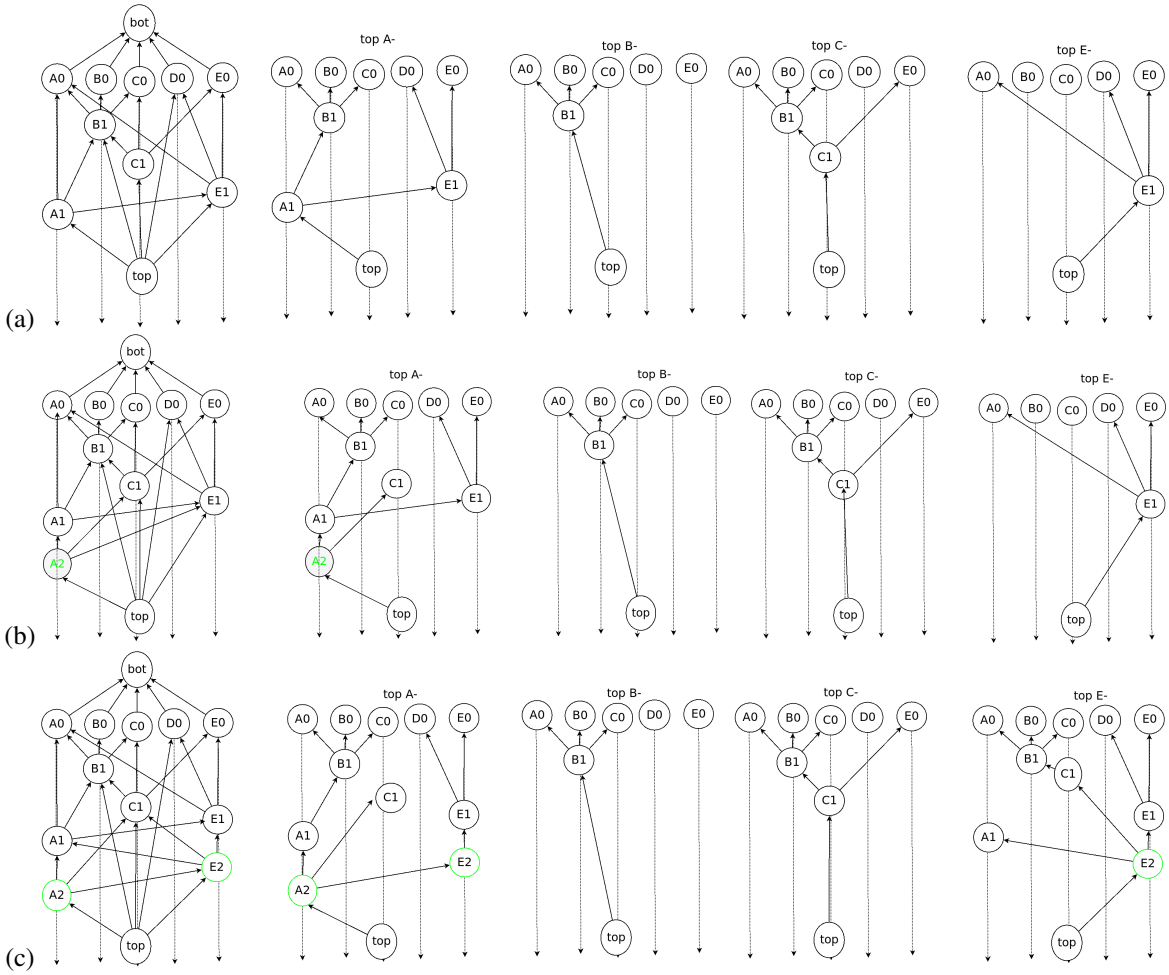

Figure 12: An example of dependency graphs on individual nodes. From (a)-(c) there is one new event block appended. There is no fork, the simplified dependency graphs become trees.

Figure 12 shows an example an dependency graphs. On each row, the left most figure shows the latest OPERA chain. The left figures on each row depict the dependency graphs of each node, which are in their compact form. When no fork presents, each of the compact dependency graphs is a tree.

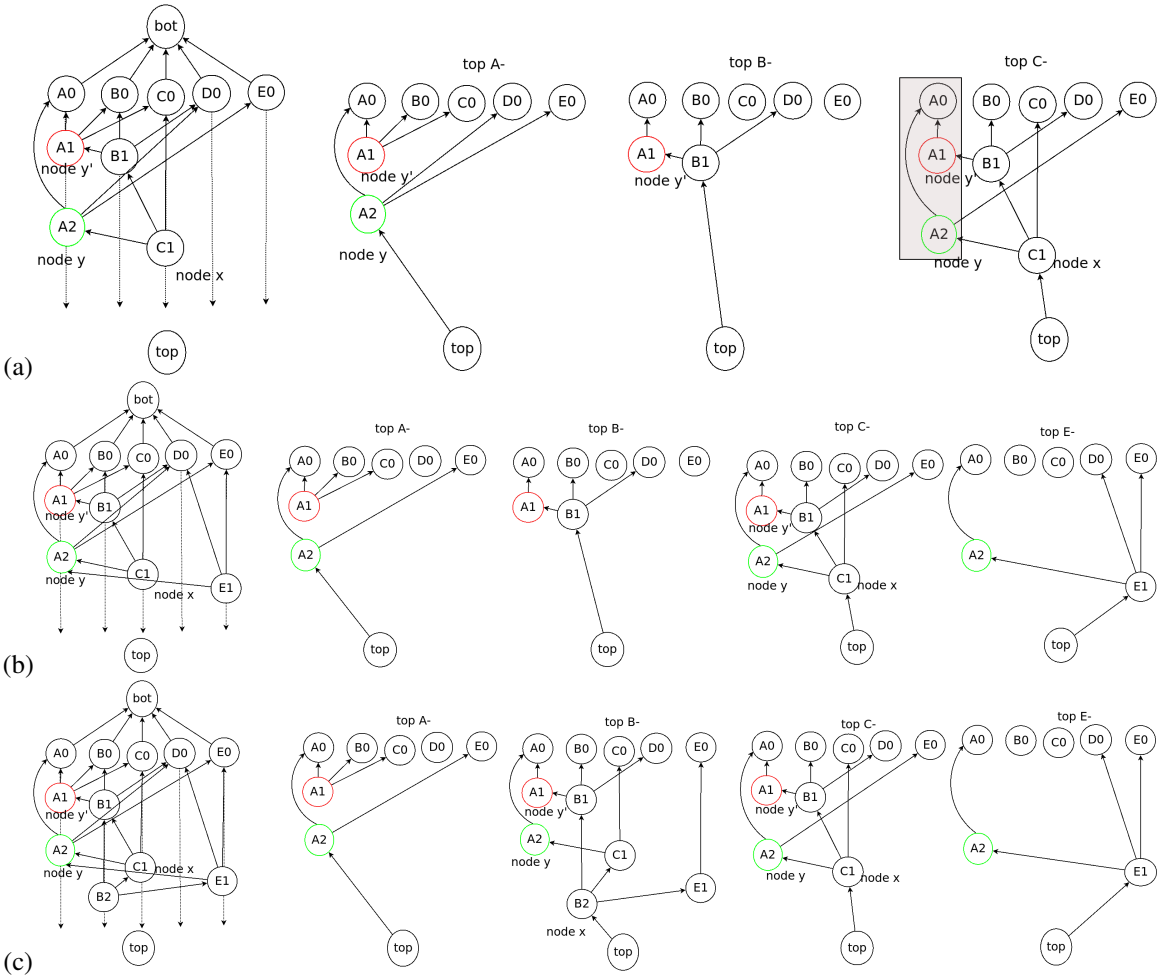

Figure 13: An example of a pair of fork events in an OPERA chain. The fork events are shown in red and green. The OPERA chains from (a) to (d) are different by adding one single event at a time.

Figure 13 shows an example of a pair of fork events. Each row shows an OPERA chain (left most) and the compact dependency graphs on each node (right). The fork events are shown in red and green vertices

3.10 Root Selection

All nodes can create event blocks and an event block can be a root when satisfying specific conditions. Not all event blocks can be roots. First, the first created event blocks are themselves roots. These leaf event blocks form the first root set R_{S_1} of the first frame f_1. If there are total 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 called 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 root set R_{S_k} is $2n/3 < |R_{S_k}| \leq n$. The event blocks including R_{S_k} before $R_{S_{k+1}}$ is in the frame f_k. The roots in $R_{S_{k+1}}$ does not belong to the frame f_k. Those 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 referenced event blocks. We apply an OR operation on each set in the flag table of the referenced event blocks.

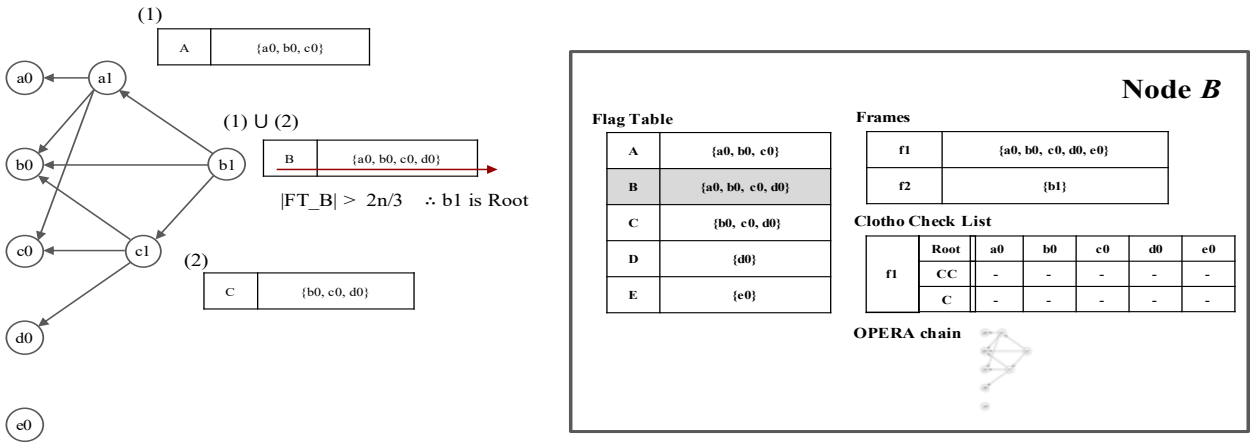

Figure 14: An Example of Root selection

Figure 14 shows an example of how to use flag tables to determine a root. In this example, b_1 is the most recently created event block. We apply an OR operation on each set of the flag tables for b_1's k referenced event blocks. The result is the flag table of b_1. If the cardinality of the root set in b_1's flag table 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 root. In this example, b_1 is added to frame f_2 since b_1 becomes new root.

The root selection algorithm is as follows:

1. The first event blocks are considered as roots.

2. When a new event block is added in the OPERA chain (DAG), we check whether the event block is a root by applying an OR operation on 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 OPERA chain, 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 $2n/3$ roots in R_{S_p}.

5. When the new root set $R_{S_{k+1}}$ is created, the event blocks from the previous root set R_{S_k} to before $R_{S_{k+1}}$ belong to the frame f_k.

3.11 Clotho Selection

A Clotho is a root that satisfies the Clotho creation conditions. Clotho creation conditions are that more than 2n/3 nodes know the root and a root knows this information.

In order for a root r in frame f_i to become a Clotho, 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}. It means that all nodes know the existence of r. If we have any root in the frame f_{i+3}, a root knows that r is spread to more than 2n/3 nodes. It satisfies Clotho creation conditions.

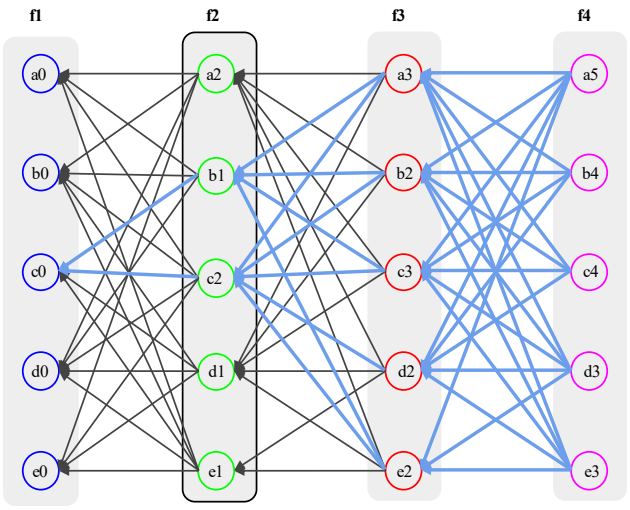

Figure 15: A verification of more than 2n/3 nodes

In the example in Figure 15, n is 5 and each circle indicates a root in a frame. Each arrow means one root can reach (happened-before) to the previous root. Each root has 4 or 5 arrows (out-degree) since 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 also can reach c_0, but we only marked b_1 and c_2 (when n is 5, more than n/3 $\geq$ 2) since we show at least more than n/3 conditions in this example. And it was marked with a blue bold arrow (Namely, the roots that can reach root c_0 have the blue bold arrow). In this situation, an event block must be able to reach b_1 or c_2 in order 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. Thus, to be a root, either must be reached). All roots in frame f_3 reach c_0 in frame f_1.

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, the root must have happened-before more than 2n/3 roots in frame f_3. Thus, the root of f_4 knows that c_0 is spread over more than 2n/3 of the entire nodes. Thus, we can select c_0 as Clotho.

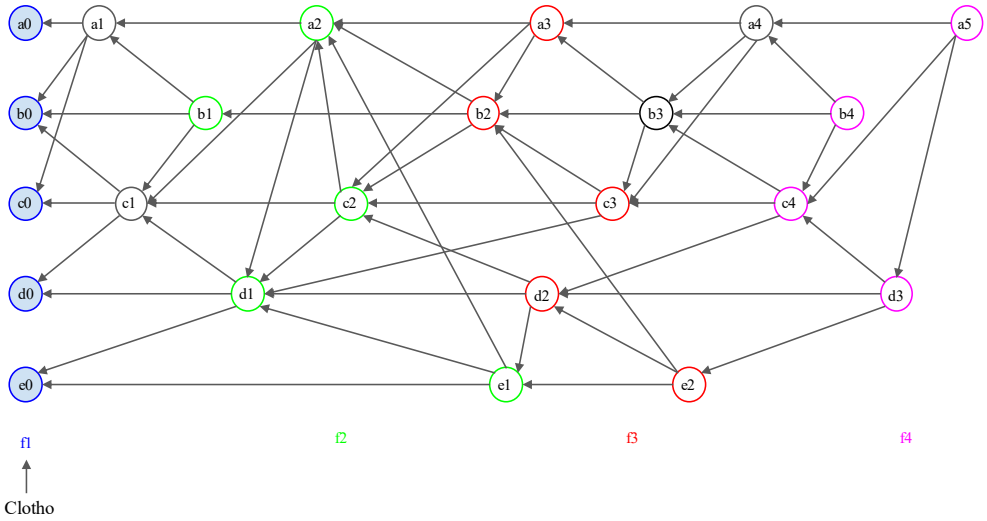

Figure 16: An Example of Clotho

Figure 16 shows an example of a Clotho. In this example, all roots in the frame f_1 have happened-before more than n/3 roots in the frame f_2. We can select all roots in the frame f_1 as Clotho since the recent frame is f_4.

Algorithm 4 Clotho Selection

```
1:  procedure CLOTHO SELECTION
2:      Input: a root r
3:      for c ∈ frame(i − 3, r) do
4:          c.is_clotho ← nil
5:          c.yes ← 0
6:          for c′ ∈ 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_clotho ← yes
```

Algorithm 4 shows the pseudo code for Clotho selection. The algorithm takes a root r as input. Line 4 and 5 set $c.is\_clotho$ and $c.yes$ to nil and 0 respectively. Line 6-8 checks whether any root c' in $frame(i-3, r)$ has happened-before with the 2n/3 condition c where i is the current frame. In line 9-10, if the number of roots in $frame(i-2, r)$ which happened-before c is more than $2n/3$, the root c is set as a Clotho. The time complexity of Algorithm 3 is $O(n^2)$, where n is the number of nodes.

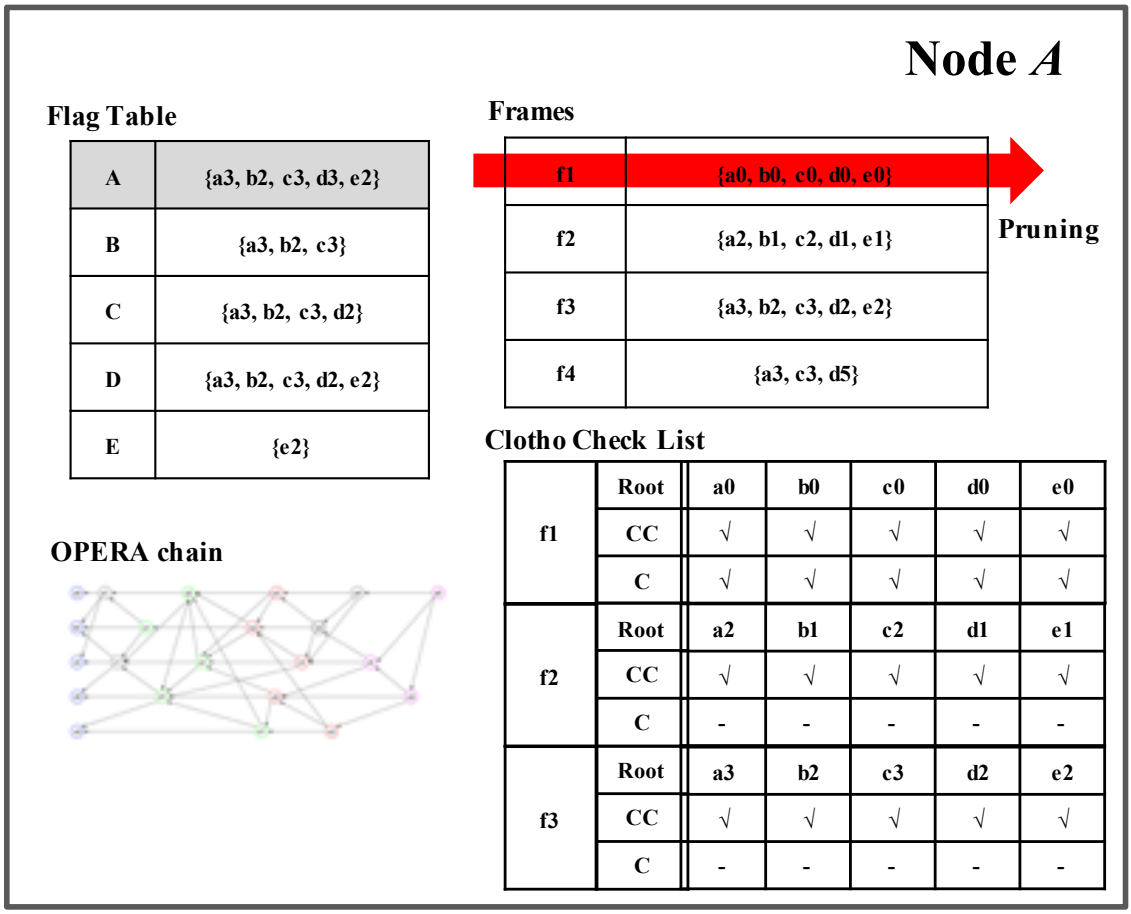

Figure 17: The node of *A* when Clotho is selected

Figure 17 shows the state of node *A* when a Clotho is selected. In this example, node *A* knows all roots in the frame f_1 become Clotho's. Node *A* prunes unnecessary information on its own structure. In this case, node *A* prunes the root set in the frame f_1 since all roots in the frame f_1 become Clotho and the Clotho Check list stores the Clotho information.

3.12 Atropos Selection

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

After a Clotho is nominated, each node then computes a candidate time of the Clotho. If there are more than two-thirds of the nodes that compute the same value for candidate time, that time value is recorded. Otherwise, each node reselects candidate time. By the reselection process, each node reaches time consensus for candidate time of Clotho as the OPERA chain (DAG) grows. The candidate time reaching the consensus is called Atropos consensus time. After Atropos consensus time is computed, the Clotho is nominated to Atropos and each node stores the hash value of Atropos and Atropos consensus time in Main-Chain (blockchain). The Main-chain is used for time order between event blocks. The proof of Atropos consensus time selection is shown in the section 5.2.

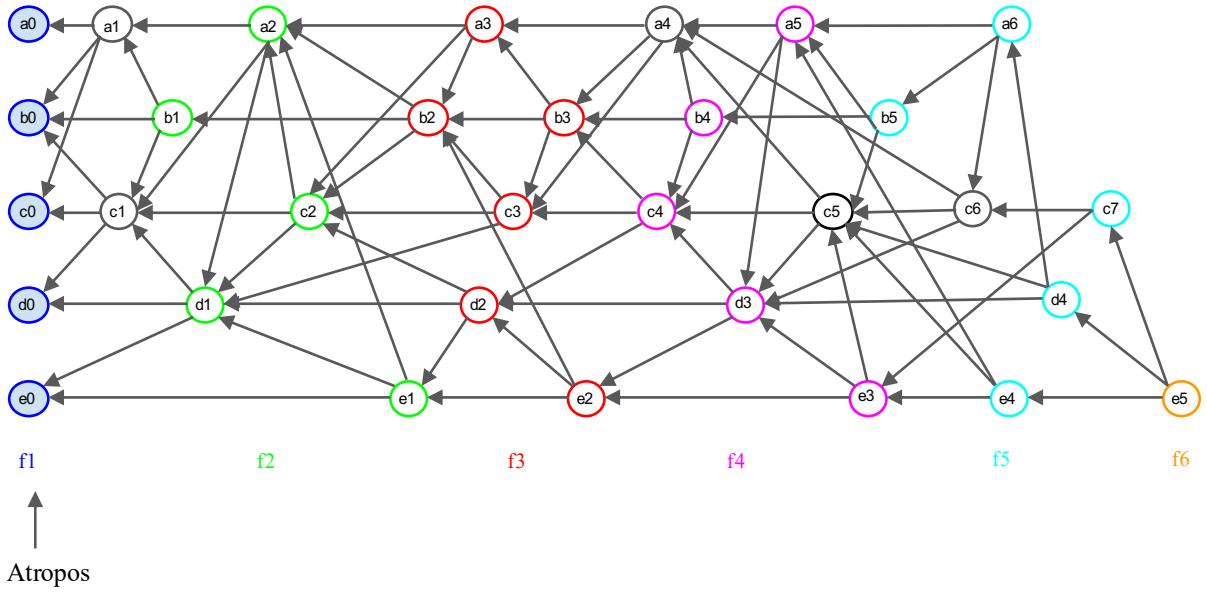

Figure 18: An Example of Atropos

Figure 18 shows the example of Atropos selection. In Figure 16, all roots in the frame f_1 are selected as Clotho through the existence of roots in the frame f_4. Each root in the frame f_5 computes candidate time using 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 of the reachable roots that f_5 takes. If r_6 knows the same candidate time than more than 2n/3, we select the candidate time as Atropos consensus time. Then all Clotho in the frame f_1 become Atropos.

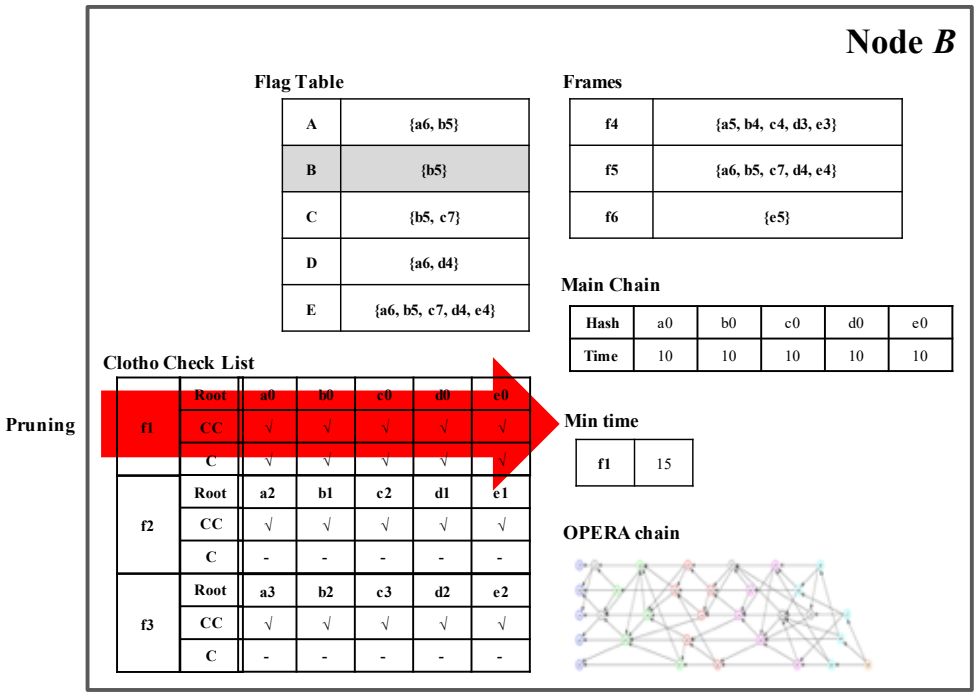

Figure 19: The node of *B* when Atropos is selected

Figure 19 shows the state of node *B* when Atropos is selected. In this example, node *B* knows all roots in the frame f_1 become Atropos. Then node *B* prunes information of the frame f_1 in clotho check list since all roots in the frame f_1 become Atropos and main chain stores Atropos information.

Algorithm 5 Atropos Consensus Time Selection

procedure ATROPOS CONSENSUS TIME SELECTION
  Input: $c.Clotho$ in frame f_i
  $c.consensus\_time \leftarrow nil$
  $m \leftarrow$ the index of the last frame f_m
  for d from 3 to (m-i) **do**
    $R \leftarrow$ be the Root set $R_{S_{i+d}}$ in frame f_{i+d}
    for $r \in R$ **do**
      if d is 3 **then**
        if r confirms c as Clotho **then**
          $r.time(c) \leftarrow r.lamport\_time$
      else if d $>$ 3 **then**
        s $\leftarrow$ the set of Root in f_{j-1} that r can be happened-before with 2n/3 condition
        t $\leftarrow$ RESELECTION(s, c)
        k $\leftarrow$ the number of root having t in s
        if d mod $h > 0$ **then**
          if $k >$ 2n/3 **then**
            $c.consensus\_time \leftarrow t$
            $r.time(c) \leftarrow t$
          else
            $r.time(c) \leftarrow t$
        else
          $r.time(c) \leftarrow$ the minimum value in s

Algorithm 6 Consensus Time Reselection

```
1: function RESELECTION
2:     Input: Root set R, and Clotho c
3:     Output: candidate time t
4:     τ ← set of all t_i = r.time(c) for all r in R
5:     D ← set of tuples (t_i, c_i) computed from τ, where c_i = count(t_i)
6:     max_count ← max(c_i)
7:     t ← infinite
8:     for tuple (t_i, c_i) ∈ D do
9:         if max_count == c_i && t_i < t then
10:            t ← t_i
11:    return t
```

Algorithm 5 and 6 show pseudo code of Atropos consensus time selection and Consensus time reselection. In Algorithm 5, at line 6, d saves the deference of relationship between root set of c and w. Thus, line 8 means that w is one of the elements in root set of the frame f_{i+3}, where the frame f_i includes c. Line 10, each root in the frame f_j selects own Lamport timestamp as candidate time of c when they confirm root c as Cltoho. In line 12, 13, and 14, s, t, and k save the set of root that w can be happened-before with 2n/3 condition c, the result of $RESELECTION$ function, and the number of root in s having t. Line 15 is checking whether there is a difference as much as h between i and j where h is a constant value for minimum selection frame. Line 16-20 is checking whether more than two-thirds of root in the frame f_{j-1} nominate the same candidate time. If two-thirds of root in the frame f_{j-1} nominate the same candidate time, the root c is assigned consensus time as t. Line 22 is minimum selection frame. In minimum selection frame, minimum value of candidate time is selected to reach byzantine agreement. Algorithm 6 operates in the middle of Algorithm 5. In Algorithm 6, input is a root set W and output is a reselected candidate time. Line 4-5 computes the frequencies of each candidate time from all the roots in W. In line 6-11, a candidate time which is smallest time that is the most nomitated. The time complexity of Algorithm 6 is $O(n)$ where n is the number of nodes. Since Algorithm 5 includes Algorithm 6, the time complexity of Algorithm 5 is $O(n^2)$ where n is the number of nodes.

In the Atropos Consensus Time Selection algorithm, nodes reach consensus agreement about candidate time of a Clotho without additional communication (i.e., exchanging candidate time) with each other. Each node communicates with each other through the Lachesis protocol, the OPERA chain of all nodes grows up into same shape. This allows each node to know the candidate time of other nodes based on its OPERA chain and reach a consensus agreement. The proof that the agreement based on OPERA chain become agreement in action is shown in the section 5.2.

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

3.13 Lachesis Consensus

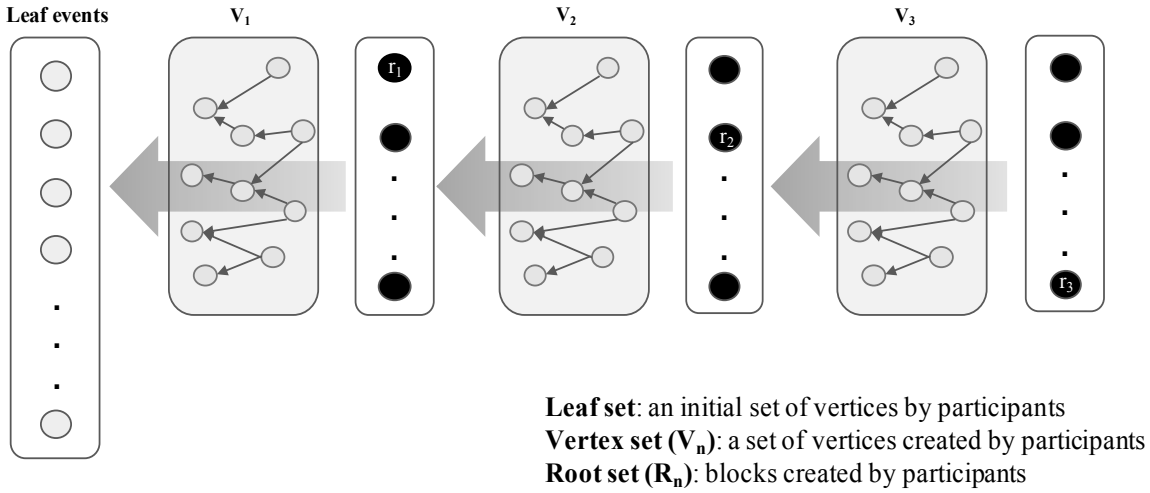

Figure 20: Consensus Method in a DAG (combines chain with consensus process of pBFT)

Figure 20 illustrates how consensus is reached through the domination relation in the OPERA chain. In the figure, 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 do not belong neither in R_{s0} nor in 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 root event blocks in root set R_{s1} and R_{s2}, respectively. r_1 is the block where a quorum or more blocks exist on a path that reaches a leaf event block. Every path from r_1 to a leaf vertex will contain 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 through blocks made by a quorum of participants. For all leaf event blocks that could be reached by r_1, they are connected with more than quorum participants through the presence of r_1. The existence of the root r_2 shows that information of r_1 is connected with more than a quorum. This kind of a path search allows the chain to reach consensus in a similar manner as the pBFT consensus processes. It is essential to keep track of the blocks satisfying the pBFT consensus process for quicker path search; our OPERA chain and Main-chain keep track of these blocks.

The sequential order of each event block is an important aspect for Byzantine fault tolerance. In order to determine the pre-and-post sequence between all event blocks, we use Atropos consensus time, Lamport timestamp algorithm and the hash value of the event block.

Topological consensus ordering

1. Smaller consensus time of Atropos, prior ordering.
2. If same consensus time, smaller Lamport timestamp, prior ordering
3. If same consensus time and same Lamport timestamp, smaller hash value, prior ordering

We assume that Atropos are **a3** and **c3**.

- If consensus time of **a3** < **c3**, **a3** has a prior ordering
- If consensus time of **b1** = **c2**, and if Lamport time of **b1** < **c2**, **b1** has a prior ordering
- If consensus time of **a2** = **b2**, and if Lamport time **a2** = **b2**, and if hash value of **a2** < **b2**, **a2** has a prior ordering

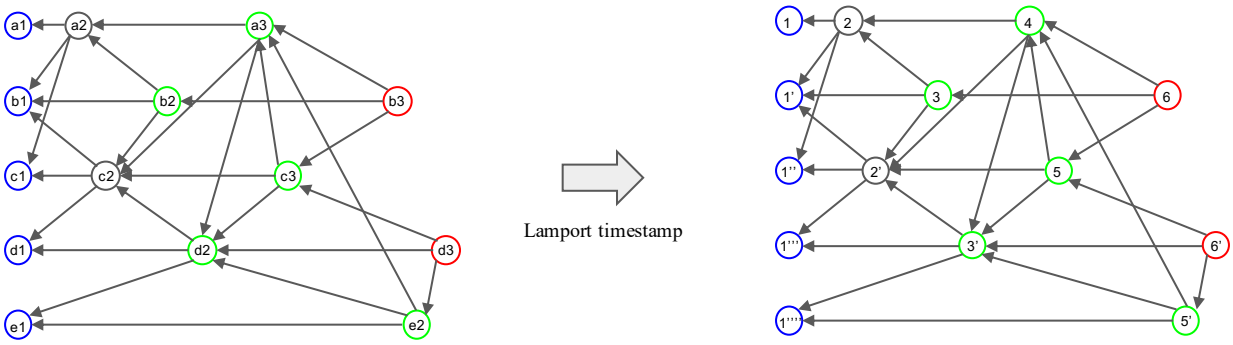

Figure 21: An example of topological consensus ordering

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

Each event block is based on the following three rules to reach an agreement:

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

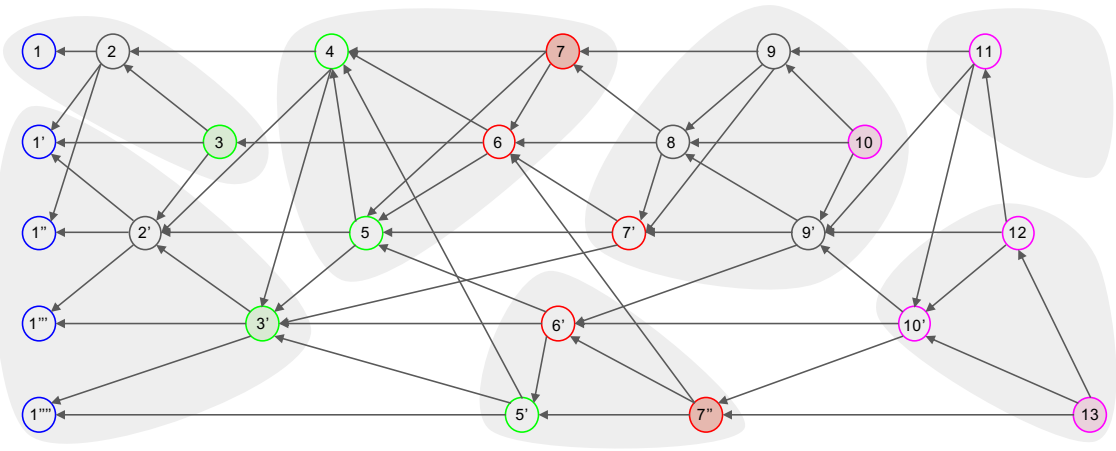

Figure 22: An Example of time ordering of event blocks in OPERA chain

Figure 22 shows the part of OPERA chain in which the final consensus order is determined based on these 3 rules. The number represented by each event block is a logical time based on Lamport timestamp. Final topological consensus order containing the event blocks are based on agreement from the apropos. Based on each Atropos, they will have different colors depending on their range.

3.14 Detecting Forks

(Fork) A 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. Denoted by $v_x \pitchfork v_y$.

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 $v_y \pitchfork v_x$.

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 definition of concurrent, we get $v_x \parallel v_y$.

Lemma 3.1. *If there is a fork $v_x \pitchfork v_y$, then v_x and v_y cannot both be roots on honest nodes.*

Here, we show a proof by contradiction. Any honest node cannot accept a fork so 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 root of n_y, where n_x and n_y are honest nodes. Since v_x is a root, it reached events created by more than 2/3 of member nodes. Similarly, v_y is a root, it reached events created by more than 2/3 of member nodes. Thus, there must be an overlap of more than n/3 members of those events in both sets. Since we assume less than n/3 members are not honest, so 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 the fork.

4 Conclusion

We further optimize the OPERA chain and Main-chain for faster consensus. By using Lamport timestamps and domination relation, the topological ordering of event blocks in OPERA chain and Main chain is more intuitive and reliable in distributed system.

We have presented a formal semantics for Lachesis protocol in Section 3. Our formal proof of pBFT for our Lachesis protocol is given in Section 5.2. Our work is the first that studies such concurrent common knowledge sematics [9] and dominator relationships in DAG-based protocols.

5 Appendix

5.1 Preliminaries

The history of a Lachesis protocol can be represented by a directed acyclic graph $G = (V, E)$, 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 left-to-right of the graph, so left vertices represent earlier events in history. A path p in G is a sequence of vertices $(v_1, v_2, \ldots, v_k)$ 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 5.1 (node)**.** *Each machine that participates in the Lachesis protocol is called a node.*

Let n denote the total number of nodes.

Definition 5.2 (event block)**.** *Each node can create event blocks, send (receive) messages to (from) other nodes.*

Definition 5.3 (vertex)**.** *An event block is a vertex of the OPERA chain.*

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 the references is self-reference, and the other k-1 references point to the top events of n_i's k-1 peer nodes.

Definition 5.4 (peer node)**.** *A node n_i has k peer nodes.*

Definition 5.5 (top event)**.** *An event v is a top event of a node n_i if there is no other event in n_i referencing v.*

Definition 5.6 (self-ref)**.** *An event v_s is called "self-ref" of event v_c, if the self-ref hash of v_c points to the event v_s. Denoted by $v_c \hookrightarrow^s v_s$.*

Definition 5.7 (ref)**.** *An event v_r is called "ref" of event v_c if the reference hash of v_c points to the event v_r. Denoted by $v_c \hookrightarrow^r v_r$.*

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

Definition 5.8 (self-ancestor)**.** *An event block v_a is self-ancestor of an event block v_c if there is a sequence of events such that $v_c \hookrightarrow^s v_1 \hookrightarrow^s \ldots \hookrightarrow^s v_m \hookrightarrow^s v_a$. Denoted by $v_c \hookrightarrow^{sa} v_a$.*

Definition 5.9 (ancestor)**.** *An event block v_a is an ancestor of an event block v_c if there is a sequence of events such that $v_c \hookrightarrow v_1 \hookrightarrow \ldots \hookrightarrow v_m \hookrightarrow v_a$. Denoted by $v_c \hookrightarrow^a v_a$.*

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

Definition 5.10 (OPERA chain)**.** *OPERA chain 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, $v_i \hookrightarrow v_j$.*

5.1.1 Domination relation

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

Definition 5.11 (pseudo top)**.** *A pseudo vertex, called top, is the parent of all top event blocks. Denoted by $\top$.*

Definition 5.12 (pseudo bottom)**.** *A pseudo vertex, called bottom, is the child of all leaf event blocks. Denoted by $\bot$.*

With the pseudo vertices, we have $\bot$ happened before all event blocks. Also all event blocks happened before $\top$. That is, for all event v_i, $\bot \rightarrow v_i$ and $v_i \rightarrow \top$.

Definition 5.13 (dom)**.** *An event v_d dominates an event v_x if every path from $\top$ to v_x must go through v_d. Denoted by $v_d \gg v_x$.*

Definition 5.14 (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. Denoted by $v_d \gg^s v_x$.*

Definition 5.15 (domfront)**.** *A vertex v_d is said "domfront" a vertex v_x if v_d dominates an immediate predecessor of v_x, but v_d does not strictly dominate v_x. Denoted by $v_d \gg^f v_x$.*

Definition 5.16 (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 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 domination.

Definition 5.17 (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 5.18 ($\frac{2}{3}$-dom set)**.** *A vertex v_d belongs to a $\frac{2}{3}$-dom set within the 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 5.1. *The $\frac{2}{3}$-dom set D_i is the same with the root set R_i, for all nodes.*

5.2 Proof of Lachesis Consensus Algorithm

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

5.2.1 Proof of Byzantine Fault Tolerance for Lachesis Consensus Algorithm

Definition 5.19 (Happened-Immediate-Before)**.** *An event block v_x is said Happened-Immediate-Before an event block v_y if v_x is a (self-) ref of v_y. Denoted by $v_x \mapsto v_y$.*

Definition 5.20 (Happened-Before)**.** *An event block v_x is said Happened-Before an event block v_y if v_x is a (self-) ancestor of v_y. Denoted by $v_x \rightarrow v_y$.*

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 followings happens: (a) $v_y \hookrightarrow^s v_x$, (b) $v_y \hookrightarrow^r v_x$, or (c) $v_y \hookrightarrow^a v_x$. We come up with the following proposition:

Proposition 5.2 (Happened-Immediate-Before OPERA)**.** $v_x \mapsto v_y$ *iff* $v_y \hookrightarrow v_x$ *iff edge* $(v_y, v_x) \in E$ *of OPERA chain.*

Lemma 5.3 (Happened-Before Lemma)**.** $v_x \rightarrow v_y$ *iff* $v_y \hookrightarrow^a v_x$.

Definition 5.21 (concurrent)**.** *Two event blocks v_x and v_y are said concurrent if neither of them happened before the other. Denoted by $v_x \parallel v_y$.*

Given two vertices v_x and v_y both contained in two OPERA chains 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 is some main definitions in Lachesis protocol.

Definition 5.22 (Leaf)**.** *The first created event block of a node is called a leaf event block.*

Definition 5.23 (Root)**.** *The leaf event block of a node is a root. When an event block v can reach more than $2n/3$ of the roots in the previous frames, v becomes a root.*

Definition 5.24 (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 5.25 (Frame)**.** *Frame f_i is a natural number that separates Root sets.*

The root set at frame f_i is denoted by R_i.

Definition 5.26 (consistent chains)**.** *OPERA chains G_1 and G_2 are consistent iff for any event v contained in both chains, $G_1[v] = G_2[v]$. Denoted by $G_1 \sim G_2$.*

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

Theorem 5.4. *All nodes have consistent OPERA chains.*

Proof. If two nodes have OPERA chains 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, so both OPERA chains 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 OPERA chains are consistent. □

Definition 5.27 (creator)**.** *If a node n_x creates an event block v, then the creator of v, denoted by $cr(v)$, is n_x.*

Definition 5.28 (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. Denoted by $v_x \pitchfork v_y$.*

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 5.5. *$v_x \pitchfork v_y$ iff $cr(v_x) = cr(v_y)$ and $v_x \parallel v_y$.*

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 definition of concurrent, we get $v_x \parallel v_y$. □

Lemma 5.6. *(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 show a proof by contradiction. Any honest node cannot accept a fork so 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 root of n_y, where n_x and n_y are honest nodes. Since v_x is a root, it reached events created by more than 2/3 of member nodes. Similary, v_y is a root, it reached events created by more than 2/3 of member nodes. Thus, there must be an overlap of more than n/3 members of those events in both sets. Since we assume less than n/3 members are not honest, so 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 the fork. This contradicts the assumption. Thus, the lemma is proved. □

Each node n_i has an OPERA chain G_i. We define a consistent chain from a sequence of OPERA chain G_i.

Definition 5.29 (consistent chain)**.** *A global consistent chain G^C is a chain if $G^C \sim G_i$ for all G_i.*

We denote $G \sqsubseteq G'$ to stand for G is a subgraph of G'.

Lemma 5.7. *$\forall G_i$ ($G^C \sqsubseteq G_i$).*

Lemma 5.8. *$\forall v \in G^C \; \forall G_i$ ($G^C[v] \sqsubseteq G_i[v]$).*

Lemma 5.9. *($\forall v_c \in G^C$) ($\forall v_p \in G_i$) (($v_p \rightarrow v_c$) $\Rightarrow v_p \in G^C$).*

Now we state the following important propositions.

Definition 5.30 (consistent root)**.** *Two chains G_1 and G_2 are root consistent, if for every v contained in both chains, and v is a root of j-th frame in G_1, then v is a root of j-th frame in G_2.*

Proposition 5.10. *If $G_1 \sim G_2$, then G_1 and G_2 are root consistent.*

Proof. By consistent chains, if $G_1 \sim G_2$ and v belongs to both chains, then $G_1[v] = G_2[v]$. We can prove the 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 $\forall u \in S$ ($u \in G_2$). Since 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 happens before v in G_2. So v belongs to f_{k+1} in G_2. The proposition is proved. □

From the above proposition, one can deduce the following:

Lemma 5.11. *G^C is root consistent with G_i for all nodes.*

Thus, all nodes have the same consistent root sets, which are the root sets in G^C. Frame numbers are consistent for all nodes.

Lemma 5.12 (consistent flag table)**.** *For any top event v in both OPERA chains 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 chains.*

Proof. From the above lemmas, the root sets of G_1 and G_2 are consistent. If v contained in G_1, and v is a root of j-th frame in G^1, then v is a root of j-th frame in G_i. Since $G_1 \sim G_2$, $G_1[v] = G_2[v]$. 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 5.31 (Clotho)**.** *A root r_k in the frame f_{a+3} can nominate a root r_a as Clotho 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 5.13. *For any root set R, all nodes nominate same root into Clotho.*

Proof. Based on Theorem 5.17, each node nominates a root into Clotho via the flag table. If all nodes have an OPERA chain with same shape, the values in flag table should be equal to each other in OPERA chain. Thus, all nodes nominate the same root into Clotho since the OPERA chain of all nodes has same shape. □

Lemma 5.14. *In the Reselection algorithm, for any Clotho, a root in OPERA chain selects the same consensus time candidate.*

Proof. Based on Theorem 5.17, if all nodes have an OPERA chain with the same partial shape, a root in OPERA chain selects the same consensus time candidate by the Reselection algorithm. □

Lemma 5.15 (Fork Lemma)**.** *If the pair of event blocks (x,y) is fork and a root has Happened-before the fork, this fork is detected in the Clotho selection process.*

Proof. We show a proof by contradiction. Assume that no node can detect the fork in the Clotho selection process.

Assume that there is a root r_i that becomes Clotho in f_i, which was selected as Clotho by n/3 of the roots in f_{i+1}. More than 2n/3 roots in f_{i+1} should have happened-before by a root in f_{i+2}. If a pair (v_x, v_y) is 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 two cases is as follows.

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

In order 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 5.23. 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} ($\because$ 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}. It 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} detect the fork in the Clotho selection of r_i, which contradicts the assumption.

It can be covered two cases. If r_i is part of the fork, we can detect in f_{i+2}. If there is fork (v_x, v_y) that can be reached by r_i, it also can be detected in f_{i+2} since we can detect the fork in the Clotho selection of r_i and it indicates that all event blocks that can be reached by r_i are detected by the roots in f_{i+2}. □

Lemma 5.16. *For a root v happened-before a fork in OPERA chain, v must see the fork before becoming Clotho.*

Proof. Suppose that a node creates two event blocks (v_x, v_y), which forms a fork. To create two Clothos that can reach both events, the event blocks should reach by more than 2n/3 nodes. Therefore, the OPERA chain can structurally detect the fork before roots become Clotho. □

Theorem 5.17 (Fork Absence)**.** *All nodes grows up into same consistent OPERA chain 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 is not equal to that in G_2. We can consider that the path difference between the nodes is a kind of fork attack. Based on Lemma 5.15, if an attacker forks an event block, each chain of G_i and G_2 can detect and remove the fork before the Clotho is generated. Thus, any two nodes have consistent OPERA chain. □

Definition 5.32 (Atropos)**.** *If the consensus time of Clotho is validated, the Clotho become an Atropos.*

Theorem 5.18. *Lachesis consensus algorithm guarantees to reach agreement for the consensus time.*

Proof. For any root set R in the frame f_i, time consensus algorithm checks whether more than 2n/3 roots in the frame f_{i-1} selects 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 Reselection process. Based on the Reselection process, the time consensus algorithm can reach agreement. However, there is a possibility that consensus time candidate does not reach agreement [10]. To solve this problem, time consensus algorithm includes minimal selection frame per next h frame. In minimal value selection algorithm, each root selects minimum value among values collected from previous root set. Thus, the consensus time reaches consensus by time consensus algorithm. □

Theorem 5.19. *If the number of reliable nodes is more than $2n/3$, event blocks created by reliable nodes must be assigned to consensus order.*

Proof. In OPERA chain, since reliable nodes try to create event blocks by communicating with every other nodes continuously, reliable nodes will 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 will be nominated as Clotho and Atropos. Thus, event block x and root y will be assigned 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, malicious node cannot try to attack after the event blocks are assigned consensus time. When the event block x has consensus time t, it cannot occur to discover new event blocks with earlier consensus time than t. There are two conditions to be assigned consensus time earlier than t for new event blocks. First, a root r in the frame f_i should be able to share new event blocks. Second, the more than 2n/3 roots in the frame f_{i+1} should be able to share r. Even if the first condition is satisfied by malicious nodes (e.g., parasite chain), the second condition cannot be satisfied since 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 be participate earlier consensus time to OPERA chain. □

5.3 Semantics of Lachesis protocol

This section gives the formal semantics of Lachesis consensus protocol. We use CCK model [9] of an asynchronous system as the base of the semantics of our Lachesis protocol. Events are ordered based on Lamport's happens-before relation. In particular, we use Lamport's theory to describe global states of an asynchronous system.

We present notations and concepts, which are important for Lachesis protocol. In several places, we adapt the notations and concepts of CCK paper to suit our Lachesis protocol.

An asynchronous system consists of the following:

Definition 5.33 (process)**.** *A 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 5.34 (channel)**.** *A 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 5.35 (state)**.** *A 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 the reference blocks exist exist before it. From a local state s^i_j, one can reconstruct a unique DAG. That is, the mapping from a local state s^i_j into a DAG is *injective* or one-to-one. Thus, for Lachesis, we can simply denote the j-th local state of a process i by the OPERA chain g^i_j (often we simply use G_i to denote the current local state of a process i).

Definition 5.36 (action)**.** *An action is a function from one local state to another local state.*

Generally speaking, an action can be either: a $send(m)$ action where m is a message, a $receive(m)$ action, and 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 Lachesis protocol, B consists of the content of an event block v. Semantics-wise, in Lachesis, there are two actions that can change a process's local state: creating a new event and receiving an event from another process.

Definition 5.37 (event)**.** *An event is a tuple $\langle s, \alpha, s'\rangle$ consisting of a state, an action, and a state.*

Sometimes, 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 5.38 (local history)**.** *A local history h_i of process i is a (possibly infinite) sequence of alternating local states — beginning with a distinguished initial state. A set H_i of 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 Lachesis protocol, we use append-only sematics. The local history may be equivalently described as either of the following:

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

$$h_i = s_0^i, s_1^i, s_2^i, s_3^i, \ldots$$

In Lachesis, a local history is equivalently expressed as:

$$h_i = g_0^i, g_1^i, g_2^i, g_3^i, \ldots$$

where g_j^i is the j-th local OPERA chain (local state) of the process i.

Definition 5.39 (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 talk about 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 5.40 (Consistent cut)**.** *A consistent cut of a run* σ *is any global state such that if* $v_x^i \rightarrow v_y^j$ *and* v_y^j *is in the global state, then* v_x^i *is also in the global state. Denoted by* $\mathbf{c}(\sigma)$.

By Theorem 5.4, all nodes have consistent local OPERA chains. The concept of consistent cut formalizes such a global state of a run. A consistent cut consists of all consistent OPERA chains. A received event block exists in the global state implies the existence of the original event block. Note that a consistent cut is simply a vector of local states; we will 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 is 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 5.41 (equivalent cuts)**.** *Two cuts* $\mathbf{c}(\sigma)$ *and* $\mathbf{c}'(\sigma')$ *are equivalent with respect 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 indexed by process identifiers. Given a fact ϕ, the modal operators are defined as follows:

Definition 5.42 (i knows ϕ)**.** $K_i(\phi)$ *represents the statement "*ϕ *is true in all possible consistent global states that include* i*'s local state".*

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

Definition 5.43 (i partially knows ϕ)**.** $P_i(\phi)$ *represents the statement "there is some consistent global state in this run that includes* i*'s local state, 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 M^C and stands for "majority concurrently knows." This is adapted from the "everyone concurrently knows" in CCK paper [9]. The definition of $M^C(\phi)$ is as follows.

Definition 5.44 (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 sometimes not feasible. Our modal operator $M^C(\phi)$ fits precisely 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 5.45 (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 that same state of knowledge, with respect to ϕ, along some cut of the run. In other words, we want a state of knowledge X satisfying: $X = M^C(\phi \wedge X)$. C^C will be defined semantically as the weakest such fixed point, namely as 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 some cut in the same asynchronous run σ including i's local state, such that ϕ is true in that cut.

Note that ϕ implies $P_i(\phi)$. But it 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 some cut $\mathbf{c}(\sigma)$. A process cannot distinguish which cut, of the perhaps many cuts that are in the run and consistent with its local state, satisfies ϕ; it can only know the existence of such a cut.

Definition 5.46 (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 5.47. *Fact ϕ is valid, denoted $\vdash \phi$, if ϕ is valid in all systems, i.e. $(\forall \Sigma)(\Sigma \vdash \phi)$.*

Definition 5.48 (local fact)**.** *A fact ϕ is local to process i in system Σ if $\Sigma \vdash (\phi \Rightarrow K_i\phi)$*

Theorem 5.20. *If ϕ is local to process i in system Σ, then $\Sigma \vdash (P_i(\phi) \Rightarrow \phi)$.*

Lemma 5.21. *If fact ϕ is local to 2/3 of the processes in a system Σ, then $\Sigma \vdash (M^C(\phi) \Rightarrow \phi)$ and furthermore $\Sigma \vdash (C^C(\phi) \Rightarrow \phi)$.*

Definition 5.49. *A fact 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, since 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 Lachesis protocol knows a fork exists (i.e., ϕ is the exsistenc of fork) in its local state s (i.e., g_j^i), then a consistent cut contains the state s will know the existence of that fork.

Definition 5.50 (i learns ϕ)**.** *Process i learns ϕ in state s_j^i of run σ if i knows ϕ in s_j^i and, for all previous states s_k^i in run σ, $k < j$, i does not know ϕ.*

The following theorem says 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 5.22 (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 have presented a formal semantics of Lachesis protocol based on the concepts and notations of concurrent common knowledge [9]. For a proof of the above theorems and lemmas in this Section, we can use similar proofs as described in the original CCK paper.

With the formal semantics of Lachesis, the theorems and lemmas described in Section 5.2 can be expressed in term of CCK. For example, one can study a fact ϕ (or some primitive proposition p) in the following forms: 'is there any existence of fork?'. One can make Lachesis-specific questions like 'is event block v a root?', 'is v a clotho?', or 'is v a atropos?'. This is a remarkable result, since we are the first that define such a formal semantics for DAG-based protocol.

6 Reference

[1] Sang-Min Choi, Jiho Park, Quan Nguyen, Kiyoung Jang, Hyunjoon Cheob, Yo-Sub Han, and Byung-Ik Ahn. Opera: Reasoning about continuous common knowledge in asynchronous distributed systems, 2018.

[2] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[3] Melanie Swan. *Blockchain: Blueprint for a new economy*. O'Reilly Media, 2015.

[4] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175, 2003.

[5] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

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

[7] Scott Nadal Sunny King. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake, 2012.

[8] Daniel Larimer. Delegated proof-of-stake (dpos), 2014.

[9] Prakash Panangaden and Kim Taylor. Concurrent common knowledge: defining agreement for asynchronous systems. *Distributed Computing*, 6(2):73–93, 1992.

[10] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.