Canopus: A Scalable and Massively Parallel Consensus Protocol
(Extended report)

Sajjad Rizvi, Bernard Wong, Srinivasan Keshav
Cheriton School of Computer Science
University of Waterloo, 200 University Ave. West, Waterloo, Ontario, Canada.

December 2, 2019

Abstract

Achieving consensus among a set of distributed entities (or *participants*) is a fundamental problem at the heart of many distributed systems. A critical problem with most consensus protocols is that they do not scale well. As the number of participants trying to achieve consensus increases, increasing network traffic can quickly overwhelm the network from topology-oblivious broadcasts, or a central coordinator for centralized consensus protocols. Thus, either achieving strong consensus is restricted to a handful of participants, or developers must resort to weaker models of consensus.

We propose Canopus, a highly-parallel consensus protocol that is 'plug-compatible' with ZooKeeper, which exploits modern data center network topology, parallelism, and consensus semantics to achieve scalability with respect to the number of participants and throughput (i.e., the number of key-value reads/writes per second). In our prototype implementation, compared to EPaxos and ZooKeeper, Canopus increases throughput by more than 4x and 16x respectively for read-heavy workloads.

1 Introduction

In the past few years, an important class of distributed applications has emerged that requires agreement on the entries of a replicated transaction log or ledger. Examples include geo-replicated database systems that support high-volume, conflict-free transaction processing [32], and private blockchains [2, 3, 1] that continuously add records to a distributed ledger. Each entry consists of a unit of computation that can read and modify the states of the application[1]. Entries are processed atomically in the order that they appear in the log or ledger.

[1]An equivalent implementation would be for each entry to store a reference and the parameters to a stored procedure.

Supporting these applications requires a consensus protocol that can scale to hundreds of nodes spanning multiple datacenters. The protocol must also be able to handle large volumes of requests, and be efficient for write-intensive workloads with nearly uniform popularity distributions.

Most commonly used consensus protocols, such as Paxos [18] and Raft [33], rely on a centralized coordinator to service client requests and replicate state changes. This introduces unavoidable latency and concentrates both processing load and network traffic at the coordinator, limiting their scalability.

Protocols such as EPaxos [28], Mencius [25], and S-Paxos [8] address scalability by moving from a single coordinator to a set of coordinators. However, as we will discuss in more detail in Section 2, message dissemination in these protocols is network topology-unaware. Their scalability is therefore still limited in wide-area deployments with restricted network link capacities. Paxos Quorum Leases [30] can improve the scalability of consensus protocols by assigning leases to non-coordinators, and serving read requests locally by lease holders. However, they are not well suited for the workloads required by our motivating applications.

In this paper, we introduce *Canopus*, a parallel, network-aware, and decentralized consensus protocol designed for large-scale deployments. It achieves scalability by (a) exploiting high performance broadcast support in modern switches and (b) parallelizing communication along an *overlay tree* [6]. Moreover, observing that modern datacenter networks use hardware redundancy, making them immune to single points of failure, and that they offer numerous disjoint paths between all pairs of servers, making network partitioning extremely rare [22, 39, 23], Canopus is optimized for the common case that an entire rack of servers never fails, and that the network is never partitioned. This greatly simplifies our design. We make three contributions:

- We present Canopus, a scalable and parallel consensus protocol.

- We provide a prototype that can be integrated with ZooKeeper to simplify deployment.

- We compare the performance of our prototype with both ZooKeeper and EPaxos. We find that our globally-distributed prototype implementation can handle up to 5 million transactions per second, and that, unlike EPaxos, the throughput increases with the number of servers participating in the consensus protocol.

2 Related Work

Many systems have been designed to solve the consensus problem. We first discuss some well-known centralized and decentralized consensus protocols. We then discuss systems that optimize performance by exploiting specialized hardware and networking technologies. We also outline related work in the area of group communication protocols and discuss the role of consensus protocols in the context of private blockchains.

2.1 Centralized Coordination

Paxos [18] is the most widely used consensus protocol. Paxos participants elect a central coordinator to enforce consensus on a single value, re-electing the coordinator if it fails. Many variants of Paxos have been proposed over the years. In Fast Paxos [20], clients send consensus requests directly to the *acceptors* to reduce the number of rounds of message exchanges. In Generalized Paxos [19], nodes can execute commutative requests out-of-order. FGCC [40] builds upon Generalized Paxos by reducing the message complexity to resolve conflicts. Although these variants aim to improve different aspects of Paxos performance, none of them significantly improve the scalability of the protocol.

Ring-Paxos [27] and Multi-Ring Paxos [26] are atomic multicast protocols developed on top of Paxos. In these protocols, nodes organize into one or more rings, and the Paxos coordinator and acceptor roles are assigned to the nodes along the ring. The major drawback of these multicast protocols is that the latency of operations increases proportionally to the number of participants.

Several other coordinator-based consensus or atomic broadcast algorithms have been proposed that have the same scalability limitations as Paxos. These include Raft [33], a consensus algorithm with understandability as its main design goal, and Zab [17], an atomic broadcast protocol used in ZooKeeper [15].

2.2 Decentralized Coordination

EPaxos [28], Mencius [25], and S-Paxos [8] have been proposed to improve the scalability of consensus algorithms. These protocols follow a decentralized approach. For example, in EPaxos and Mencius, each node serves as the coordinator for the clients connected to that node. The nodes then forward requests to each other for conflict resolution and ordering. S-Paxos distributes the message dissemination load across nodes in a similar way. However, the nodes send the request IDs to a centralized coordinator that runs the Paxos algorithm to order the requests.

Two primary limitations restrict the scalability of these systems. First, they are not network topology-aware. For example, each node broadcasts requests to every other participating node, or at least to a quorum of the nodes. Thus, multiple instances of the same request and response may traverse one or more oversubscribed intra-data-center or high-latency wide-area links, which limits throughput. In contrast, Canopus organizes the nodes into a network-aware overlay tree, and remote results are retrieved *once* then shared among peers. Second, these protocols broadcast *both* read and write requests. Canopus does not broadcast read requests. Instead, Canopus introduces small delays to the read operations that ensure that read operations are correctly linearized with respect to any concurrent writes operations.

The Giraffe consensus protocol logically organizes nodes into an interior-node-disjoint forest [38]. It uses a variant of Paxos to order write requests within a tree and across trees in the forest. Unlike Canopus, Giraffe does not provide linearizable consistency, and each tree in Giraffe has its own coordinator, which limits its scalability.

Canopus is similar to the recently proposed AllConcur [34] consensus algorithm, which organizes nodes in the form of an overlay digraph. In each consensus round, each AllConcur node atomically broadcasts a message to its successors in the graph while keeping track of node failures. AllConcur nodes are expected to be part of the same InfiniBand network, typically within a single datacenter. In comparison, Canopus is a network-aware protocol that organizes the nodes in the form of an wide-area overlay tree. Nodes located in the same rack form a virtual group, which can use any reliable broadcast or stronger protocol to reach agreement. Canopus efficiently stitches together these virtual groups, as described later in this paper.

2.3 Consensus Exploiting Network Hardware

Several recent systems [21, 35, 10, 16, 9] exploit fast networking devices, e.g., low latency switches, or specialized hardware, such as FPGAs, to achieve consensus with high throughput and low latency. Since they require extensive in-network support, these systems can only be deployed within the same rack or the same datacenter. In contrast, Canopus is designed to provide consensus between nodes across globally-distributed datacenters. Network-hardware-aware consensus protocols are compatible with Canopus, in that these systems can be used to achieve consensus among nodes within a Canopus virtual group.

2.4 Group Communication Protocols

Many overlay and group communication protocols have been proposed in the research literature [24, 36]. These protocols efficiently and reliably disseminate messages across a large number of participants. Canopus uses the LOT group communication protocol [6], which is similar to Astrolabe [41], to disseminate protocol state. LOT embodies three insights to achieve high performance: (1) communication should be *topology aware*; most communication should occur between topologically-close nodes, (2) communication should be *hierarchical*, so that message dissemination to n nodes takes $O(log(n))$ steps, and (3) communication should be *parallel*, so that all nodes can make progress independent of each other.

2.5 Private Blockchains

Private blockchains [2, 3, 1] can be viewed as a distributed append-only ledger that is replicated across a large number of nodes. Blockchains can be used for many different applications such as cryptocurrencies, stocks settlements, fast currency conversions, and smart contracts. A common problem with blockchains is poor scalability, which is primarily due to the use of proof-of-work or topologically-unaware consensus protocols. Blockchains use consensus to resolve conflicts among concurrent append requests to the next ledger index. However, the throughputs of these protocols peak at a few tens of thousands of transactions/second with tens of participants [42]. Canopus can be used as a scalable consensus protocol for private blockchains to significantly increase their throughput. Although the current design of Canopus does not provide Byzantine fault tolerance, we are integrating Byzantine fault-tolerance into Canopus in ongoing work.

2.6 Scalability of Read Requests

Several read optimizations have been proposed to reduce the latency of read requests for read-intensive workloads. Paxos Quorum Leases [30] (PQL) grants read leases to a set of objects to nodes that frequently access them. Lease holders can then serve read requests locally as long as the lease is valid. PQL performs well for workloads where data popularity is highly skewed. In Megastore [7], nodes can read data locally if the local-replica is the most up-to-date. Otherwise, nodes read from a majority of replicas. Canopus include an optional read optimization (§7.2) that allows the nodes to always read from the local replica.

3 System Assumptions

We now specify our six design assumptions:

Network topology: Canopus makes only two minor assumptions about the nature of its data center network. These are likely to hold for the vast majority of modern data center networks:

1. **All machines running Canopus in the same datacenter are hosted in the same rack, and are dual-connected to Top-of-Rack (ToR) switches** that provide low-latency connectivity to all the machines in its rack. If the number of Canopus nodes in a datacenter exceeds what can be hosted in a single rack, we assume that we have control over the physical location of the Canopus' nodes within a datacenter, including the location in the racks or whether Canopus runs in a VM or a physical machine.

2. **ToR switches are connected to each other through a hierarchical network fabric**. In general, we expect latency to increase and bandwidth to decrease as hop count between the pair of communicating nodes increases.

Although Canopus will perform correctly (i.e., either achieve consensus or stall) without the next two assumptions, for acceptable performance, we also assume that:

3. **Full rack failures are rare.** More precisely, we assume that although individual machines, ToR switches, and/or links may fail, it is rarely the case that all machines in a rack are simultaneously unreachable from off-rack machines. Single-failure fault tolerance is easily achieved by using dual-NIC machines connected to two ToR switches on each rack. In the rare cases of rack failures, Canopus halts until the racks have recovered. Moreover, rack

failures do not cause the system to enter into an unrecoverable state. This is because the nodes persistently store their protocol state, and our system can resume when enough nodes in the failed racks recover.

4. **Network partitions are rare.** A network partition within a rack requires both ToR switches to simultaneously fail, which we assume is rare. Similarly, we assume that the network provides multiple disjoint paths between ToR switches, so that the ToR switches are mutually reachable despite a link or switch failure. Finally, we assume that there are multiple redundant links between data centers, so that a wide-area link failure does not cause the network to partition.

Many modern datacenters use network topologies that meet these two assumptions [39, 11], including Clos-like topologies such as fat-tree [5], VL2 [14], and leaf/spine networks. These failure assumptions simplify the design of Canopus and allow us to optimize Canopus for the common case.

In the unlikely event of a network partition, we believe it is reasonable for a consensus protocol to stall while waiting for the network to recover, resuming seamlessly once the network recovers. Indeed, this is the only reasonable response to a network partition where a majority of consensus participants do not reside in any single partition. We believe that, given both the rarity of network partitions and the severity in which they affect applications and services, it is not worthwhile for a consensus protocol to provide limited availability at the expense of a significant increase in protocol complexity. Planned shutdown of a rack is possible and does not result in any loss in availability.

5. Client-coordinator communication: In Canopus, as in many other consensus systems, clients send key-value read and write requests to a Canopus *node*. We assume that a client that has a pending request with a particular node does not send a request to any other node. This is to allow serialization of client requests at each node. If a client were to concurrently send a write and a read request to two different nodes, the read request could return before the write request even if the read was sent after the write, which would violate program order.

6. Crash-stop failures: We assume that nodes fail by crashing and require a failed node to rejoin the system using a join protocol. The Canopus join protocol closely resembles the join protocol for other consensus systems [33]. Canopus detects node failures by using a method similar to the heartbeat mechanism in Raft.

4 Canopus

Canopus is a scalable distributed consensus protocol that ensures live nodes in a Canopus group agree on the same ordered sequence of operations. Unlike most previous consensus protocols, Canopus does not have a single leader and uses a virtual tree overlay for message dissemination to limit network traffic across oversubscribed links. It leverages hardware redundancies, both within a rack and inside the network fabric, to reduce both protocol complexity and communication overhead. These design decisions enable Canopus to support large deployments without significant performance penalties.

The Canopus protocol divides execution into a sequence of *consensus cycles*. Each cycle is labelled with a monotonically increasing cycle ID. During a consensus cycle, the protocol determines the order of pending write requests received by nodes from clients before the start of the cycle and performs the write requests in the same order at every node in the group. Read requests are responded to by the node receiving it. Section 5 will describe how Canopus provides linearizable consistency while allowing any node to service read requests and without needing to disseminate read requests.

Canopus determines the ordering of the write requests by having each node, for each cycle, independently choose a large random number, then ordering write requests based on the random numbers. Ties are expected to be rare and are broken deterministically using the unique IDs of the nodes. Requests received by the same node are ordered by their order of arrival, which maintains request order for a client that sends multiple outstanding requests during the same consensus cycle.

4.1 Leaf-Only Trees

During each consensus cycle, each Canopus node disseminates the write requests it receives during the previous cycle to every other node in a series of *rounds*. For now, we will assume that all nodes start the same cycle in parallel at approximately the same time. We relax this assumption in Section 4.4, where we discuss how Canopus nodes self-synchronize.

Instead of directly broadcasting requests to every node in the group, which can create significant strain on oversubscribed links in a datacenter network or wide-area links in a multi-datacenter deployment, message dissemination follows paths on a topology-aware virtual tree overlay.

Specifically, Canopus uses a Leaf-Only Tree overlay [6], that allows nodes arranged in a logical tree to compute an arbitrary global aggregation function. We adapt LOT for use in a consensus protocol; from this point on, we will always be referring to our modified

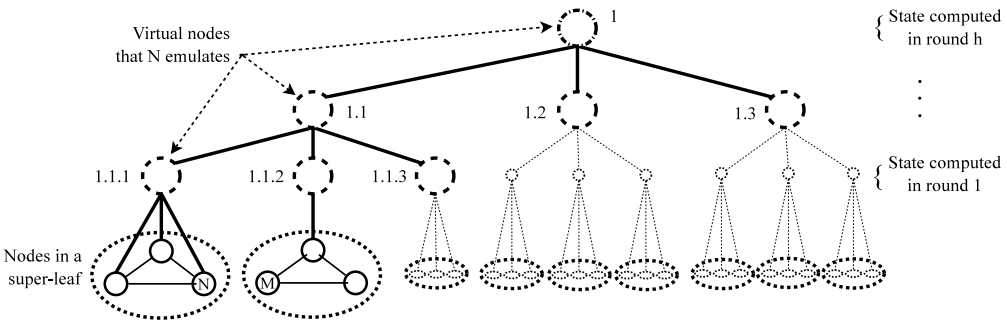


Figure 1: An example of a leaf-only tree (LOT). Only the leaf nodes exist physically and the internal nodes are virtual. A leaf node emulates all of its ancestor virtual nodes.

version of LOT.

A LOT has three distinguishing properties:

i. Physical and virtual nodes: In LOT, only the leaf-nodes exist physically (in the form of a dedicated process running in a physical machine). Internal nodes are virtual and do not exist as a dedicated process. Instead, each leaf node emulates *all of its ancestor nodes*. For clarity, we denote a leaf node as a *pnode* and an internal node as a *vnode*.

ii. Node emulation: Each pnode emulates all of its ancestor vnodes. i.e., each pnode is aware of and maintains the state corresponding to all of its ancestor vnodes. Thus, the current state of a vnode can be obtained by querying any one of its descendants, making vnodes inherently fault tolerant, and making vnode state access parallelizable.

iii. Super-leaves: All the pnodes located within the same rack are grouped into a single *super-leaf* for two reasons. First, this reduces the number of messages exchanged between any two super-leaves; instead of all-to-all communication between all the pnodes in the respective super-leaves, only a subset of the super-leaf nodes, called its *representatives*, communicate with another super-leaf on behalf of their peers. Second, because all the pnodes in a super-leaf replicate their common parents' state, a majority of the super-leaf members need to simultaneously fail to cause the super-leaf to fail.

Figure 1 shows an example of a leaf-only tree consisting of 27 pnodes. There are three pnodes per super-leaf. The pnode N emulates all of its ancestor vnodes `1.1.1`, `1.1`, and `1`. The root node `1` is emulated by all of the pnodes in the tree.

We assume that during Canopus initialization, each node is provided with the identities and IP addresses of the peers in its own super-leaf.

4.2 Consensus Cycle

Each consensus cycle consists of h *rounds* where h is the height of the LOT. Each consensus cycle is labelled by a monotone non-decreasing *cycle ID*. Canopus ensures that nodes concurrently execute the same consensus cycle, which we further explain in Section 4.4. In the following sections, we describe the details of the messages exchanged in each round of a consensus cycle.

First round in a consensus cycle: In the first round of a consensus cycle, each pnode prepares a *proposal message* and broadcasts the message to the peers in its super-leaf using a reliable broadcast protocol, which we describe in detail in Section 4.3. A proposal message contains the set of pending client requests, group membership updates, and a large random number generated by each node at the start of the consensus cycle. This random number, called the *proposal number*, is used to order the proposals across the nodes in each round in the consensus cycle. A proposal message is also tagged with the cycle ID, the round number, and the vnode ID whose state is being represented by the proposal message. The group membership updates include the membership changes that happened before starting the current consensus cycle. Note that proposal messages sent in the first round are used to inform the receivers about the start of the next consensus cycle.

After receiving the round-1 proposal messages from all the peers in its super-leaf, each pnode independently orders the proposals according to their (random) proposal numbers. The sorted list of proposals represents the partial order of the requests in the current round and constitutes the state of the parent vnode of the super-leaf. A node finishes its first round after it has obtained proposal messages from all its super-leaf peers, and then immediately starts the next round in the consensus cycle.

Round-i in a consensus cycle: In each subsequent round, each node independently prepares a new proposal message, which is used to share the computed state

from the previous round with other super-leaves. The list of requests in the proposal message is simply the sorted list of requests obtained in the previous round. Similarly, the proposal number in the new message is the largest proposal number from the previous round. Thus, a proposal message for round i can be defined as $M_i = \{R'_{i-1}, N'_{i-1}, F'_{i-1}, C, i, v\}$, where R' is the sorted list of proposals from the previous round, N'_{i-1} is the largest proposal number in the previous round, F'_{i-1} is the set of membership updates received in the proposals from the previous round, C is the cycle ID, i is the round number, and v is the vnode ID, or the pnode ID in the case of the first round.

In round i, the pnodes compute, in parallel, the state of their height-i ancestor vnode. For example, the root vnode in Figure 1 has a height of 3, and therefore, its state is computed in the third round. As the current state of a vnode is just the merged and sorted list of proposals belonging to its child pnodes or vnodes, each super-leaf independently gathers the proposals of that vnode's children from their `emulators`, where a vnode's emulator is a pnode that is known to have the vnode's state; see § 4.6 for details.

To obtain the proposal belonging to a vnode, representatives from each super-leaf select one of that vnode's emulators and sends a proposal-request message to the selected emulator. For instance, to compute the state of the node `1.1`, representative node N in Figure 1 sends a proposal-request to emulators of the vnodes `1.1.2` and `1.1.3` – it already knows the state of the vnode `1.1.1`, which was computed in the previous round.

The proposal-request message contains the cycle ID, round number, and the vnode ID for which the state is required. In response to receiving a proposal-request message, the receiver pnode replies with the proposal message belonging to the required vnode. If the receiver pnode has not yet computed the state of the vnode, it buffers the request message and replies with the proposal message only after computing the state of the vnode. When the super-leaf representative receives the proposal message, it broadcasts the proposal to its super-leaf peers using reliable broadcast. These peers then independently sort the proposals, resulting in each node computing the state of their height-i ancestor. Note that representatives from all super-leaves issue proposal-requests in parallel, and all nodes in all super-leaves independently and in parallel compute the state of their height-i ancestor.

Completing a consensus cycle: The consensus protocol finishes after all nodes have computed the state of the root node, which takes time proportional to the logarithm of the number of Canopus pnodes. At this point, every node has a total order of requests received by all the nodes. The node logs the requests in its request log for execution. It then initiates the next consensus cycle with its super-leaf peers if it received one or more client requests during the prior consensus cycle.

4.3 Reliable-Broadcast in a Super-Leaf

The nodes in a super-leaf use reliable broadcast to exchange proposal messages with each other. For ToR switches that support hardware-assisted atomic broadcast, nodes in a super-leaf can use this functionality to efficiently and safely distribute proposal messages within the super-leaf.

If hardware support is not available, we use a variant of Raft [33] to perform reliable broadcast. Each node in a super-leaf creates its own dedicated Raft group and becomes the initial *leader* of the group. All other nodes in the super-leaf participate as *followers* in the group. Each node broadcasts its proposal messages to the other nodes in its super-leaf using the Raft log replication protocol in its own Raft group. If a node fails, the other nodes detect that the leader of the group has failed, and elect a new leader for the group using the Raft election protocol. The new leader completes any incomplete log replication, after which all the nodes leave that group to eliminate the group from the super-leaf.

Reliable broadcast requires $2F + 1$ replicas to support F failures. If more than F nodes fail in the same super-leaf, the entire super-leaf fails and the consensus process halts.

4.4 Self-Synchronization

So far we have assumed that all of the nodes start at the same consensus cycle in parallel. Here, we describe how the nodes are self-synchronized to concurrently execute the same consensus cycle.

Before starting a consensus cycle, a node remains in an idle state. In this state, its network traffic is limited to periodic heartbeat messages to maintain its liveness. A new consensus cycle only starts when the node receives outside prompting. The simplest version of this prompting is when the node receives a client request for consensus. A client request triggers the node to start a new consensus cycle and broadcast a proposal message to other peers in its super-leaf. The proposal message informs the peers that it is time to start the next consensus cycle, if it is not already under way. The representatives, in effect, send proposal-request messages to other super-leaves that trigger them to start the next consensus cycle, if they have not started yet.

Alternatively, a node, while in the idle state, may receive either a proposal message from its peers in its super-leaf, or a proposal-request message from another super-leaf. This indicates that a new cycle has started,

and therefore, this node begins the next consensus cycle. In this scenario, the node's proposal message contains an empty list of client requests. As a result, the nodes are self-clocked to execute the same consensus cycle.

4.5 Super-Leaf Representatives

A super-leaf representative fetches the state of a required vnode on behalf of its super-leaf peers and broadcasts the fetched state to the peers using reliable broadcast (§ 4.3). We use a variant of Raft's leader election and heartbeat protocols [33] to allow nodes in a super-leaf to select one or more super-leaf representatives and detect representative failures. A representative failure triggers a new election to select a replacement representative.

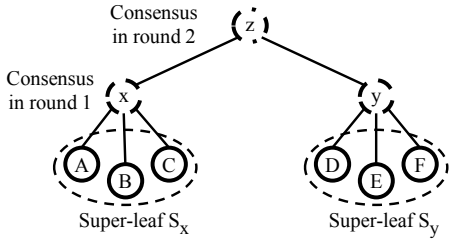
The representatives of a super-leaf are each individually responsible for fetching the states of the tree's vnodes. A representative can fetch the states of more than one vnode. However, to improve load balancing, we recommend that different representatives fetch the states of different vnodes. The vnode to representative assignment can be performed deterministically by taking the modulo of each vnode ID by the number of representatives in a super-leaf. Because representatives of a super-leaf are numbered and ordered (e.g., representative-0, representative-1, etc.), the result of the modulo operation can be used to determine which representative is assigned to a vnode without the need for any communication or consensus.

4.6 Emulation Table

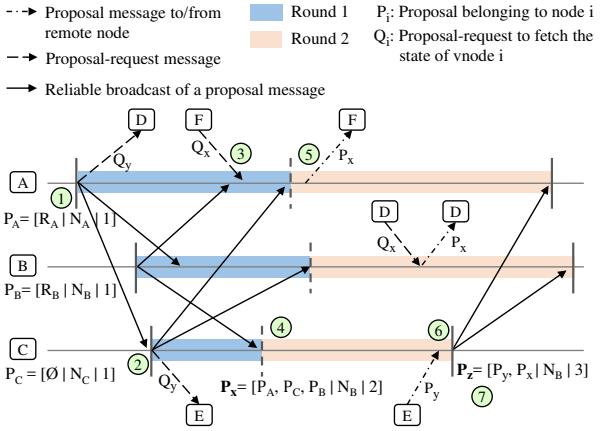
Thus far, we have not discussed how Canopus nodes actually communicate with other nodes. To initiate communication with an emulator of a vnode (a pnode in the sub-tree of the vnode) with a given ID, LOT maintains an *emulation table* that maps each vnode to a set of IP addresses of the pnodes that emulate that vnode. For example, for the LOT given in Figure 1, the table would map vnode 1.1 to the set of IP addresses of the nine pnodes that emulate this vnode.

We assume that at system initialization time, the emulation table is complete, correct, and made available to all pnodes. Subsequently, failed pnodes must be removed from the emulation tables maintained by all non-failed pnodes. Note that this itself requires consensus. Hence we maintain emulation tables *in parallel* with the consensus process, by piggybacking changes in the emulation table on proposal messages.

Specifically, in the first round of a consensus cycle, pnodes list membership changes (determined based on an within-super-leaf failure-detector protocol, such as through Raft heartbeats) in their proposal messages.



(a) Six nodes are arranged in two super-leaves in a LOT of height 2.



(b) Timing diagram of the events occurring in the super-leaf S_x, which consists of the nodes A, B, and C.

Figure 2: Illustration of a consensus cycle in Canopus.

Membership changes for a pnode include the list of pnodes that joined or left its super-leaf before the pnode started its current consensus cycle. At the end of the consensus cycle, all the pnodes have the same set of proposal messages, and thus the same set of membership changes are delivered to each pnode. Therefore, each pnode applies the same membership updates to the emulation table at the end of each consensus cycle. As a result, each pnode has the same emulation table and same membership view of LOT in each consensus cycle.

4.7 An Illustrative Example

We illustrate Canopus using an example in which six nodes are arranged in two super-leaves in a LOT of height 2 as shown in Figure 2(a). Nodes A, B, and C comprise super-leaf S_x, and the nodes D, E, and F comprise the super-leaf S_y. The nodes in S_x and S_y are connected with the common vnodes x and y respectively. The vnodes x and y are connected with the root vnode z. In the first round, the states of vnodes x and y are computed, which establishes consensus within the super-leaves. In the second round, the state of the root vnode z is computed, which establishes consensus between super-leaves. Figure 2(b) shows the events occur-

ring in the super-leaf S_x during the consensus cycle. The events are labelled in circles and explained as follows:

1. The consensus cycle starts: Nodes A and B start the first round of the consensus cycle. At the start of the cycle, nodes A and B have the pending requests R_A and R_B. The nodes prepare proposal-messages and broadcast them to their peers in the super-leaf.

Assume that the proposals of nodes A, B, and C are $P_A = \{R_A \mid N_A \mid 1\}$, $P_B = \{R_B \mid N_B \mid 1\}$, and $P_C = \{\phi \mid N_C \mid 1\}$, where R_i is the ordered set of pending requests on node i, N_i is the proposal number, and 1 is the round number. We omit the other information from the proposals for simplicity.

2. A proposal-request message is sent: Node C receives the proposal message from node A and starts its consensus cycle. Assuming that nodes A and C are the representatives for super-leaf S_x, these nodes send proposal-request messages Q_y to D and E respectively to redundantly fetch the state of vnode y. The nodes in S_x require the state of y to compute the state of the ancestor vnode z in the next round.

3. A proposal-request is received for an unfinished round: Node A receives a proposal-request message from F, which belongs to the next round of the current consensus cycle, asking for the proposal P_x. As node A has not yet finished its first round that results in P_x, it buffers the request and does not send any reply until it finishes its first round.

4. The first round is completed: Node C receives the proposal message P_B from B. As node C has received the proposals from all the peers in its super-leaf, it finishes its first round. Node C sorts the requests in the received proposals and its own proposal according the proposal numbers. The sorted list of requests is the consensus result from the first round, and comprises the current state of the vnode x. When nodes A and B finish their first round, they also have the same sorted list of requests.

To continue with the next round, the nodes prepare the proposal-messages that contain the sorted list of requests and the largest proposal number from the last round. In this example, the three nodes prepare the round-2 proposal $P_x = \{P_A, P_C, P_B \mid N_B \mid 2\}$, assuming $N_A < N_C < N_B$.

5. A proposal-request is served: Node A continues in the next round after finishing the first round and prepares the proposal P_x. It replies to the pending proposal-request Q_x, which it received from node F.

6. Coordinating the proposals: Node C receives the reply of its request Q_y for the round-2 proposal P_y, and reliably broadcasts P_y to other peers in its super-leaf. Assume that $P_y = \{P_D, P_E, P_F \mid N_F \mid 2\}$.

7. The consensus cycle is completed: After receiving all the required round-2 proposals, node C finishes the second round of the consensus cycle. At the end of the round, node C has the proposal $P_z = \{P_y, P_x \mid N_B \mid 3\}$, assuming $N_F < N_B$. As node C has now finished calculating the state of the root node, it finishes the consensus cycle. At the end of the cycle, node C has the consensus on the requests, which is $\{R_D, R_E, R_F, R_A, R_C, R_B\}$. Other nodes will have the same ordering of the requests when their second round is finished. Node C applies the requests in its copy of the commit log. Note that all node eventually agree on identical copies of this log.

5 Linearizability

Canopus provides linearizability by totally ordering both read and update requests. Interestingly, Canopus enforces global total order *without* disseminating read requests to participating nodes, significantly reducing network utilization for read-heavy workloads. Instead, Canopus nodes delay read requests until all concurrently-received update requests are ordered through the consensus process. Once the update request ordering is known, each Canopus node locally orders its pending read requests with respect to its own update requests such that the request orders of its clients are not violated. This trades off read latency for network bandwidth.

During consensus cycle C_i, the nodes buffer the requests received from the clients. The requests accumulated on a node N_i during cycle C_j form a request set S_i^j. In Canopus, the requests within a request set are sequential and follow the receive order. In contrast, the request sets across the nodes in cycle C_j, which we denote as S_*^j, are concurrent. To order the concurrent sets in S_*^j, Canopus executes the next instance C_{j+1} of the consensus protocol. The consensus process gives a total order of the concurrent sets in S_*^j. The total order of the concurrent sets translates to the linearized order of the requests received during the cycle C_j. Note that the requests in a request set are never separated. We are instead ordering the request sets to determine a total order of requests. At the end of the consensus process, each node has the same total order of write requests, and inserts its read requests in the appropriate positions to preserve its own request set order.

In this approach, each node delays its read requests for either one or two consensus cycles to provide linearizability. If a request is received immediately after starting a consensus cycle C_j, the read request is delayed for two consensus cycles: The currently executing cycle C_j, and the next cycle C_{j+1} in which the request sets S_i^j are ordered and executed. If the request is received just before completing the currently executing cycle C_j, the

read is delayed for only one consensus cycle C_{j+1}.

6 Correctness Properties

Canopus provides the following properties, which are similar to those provided by Paxos and EPaxos:

Nontriviality: A node commits and serves only those requests that it receives from clients.

Agreement: All correct nodes that complete consensus cycle c_i commit the same ordered-set of requests at the end of c_i. If a correct node commits a request r_a before a request r_b at the end of cycle c_i, all correct nodes commit r_a before r_b at the end of cycle c_i.

Liveness: If a super-leaf has not failed, the live nodes can eventually achieve consensus after receiving all the messages required to complete the current consensus cycle. Stronger liveness property cannot be guaranteed due to FLP impossibility result [12].

In addition to the above mentioned safety and liveness properties, Canopus also guarantees that:

Linearizability: All the correct nodes observe the same order of reads and write requests.

FIFO order of client requests: For each client, a node serves requests in the order it received them from the client. If a node receives a request r_a before r_b, it executes r_a before r_b, and it replies r_a before r_b.

THEOREM 1: For a height-h LOT consisting of n super-leaves, all the live descendant-nodes $\widehat{E}^r_w$ of the root-vnode w that eventually complete the last round $r = h$ have the same ordered-set of messages.

$$\underset{i,j \in \widehat{E}^r_w}{\forall} \quad M^r_i = M^r_j \tag{1}$$

where M^r_i is the set of messages that a live node i has after completing the round r of the current consensus cycle.

We provide a proof sketch for the special case of the first consensus cycle for height-2 tree with n super-leaves. The complete proof is given in Appendix A.

The proof assumes that:

A1 All nodes are initialized with the same emulation table and membership view.

A2 The network cannot be partitioned, messages are not corrupted, messages are eventually delivered to a live receiver, and that nodes fail by crashing. Node failures within a super-leaf are detected by using heartbeats and timeouts similar to Raft.

A3 The super-leaf-level structure of LOT does not change: Nodes may leave or join super-leaves, however, super-leaves are not added or removed.

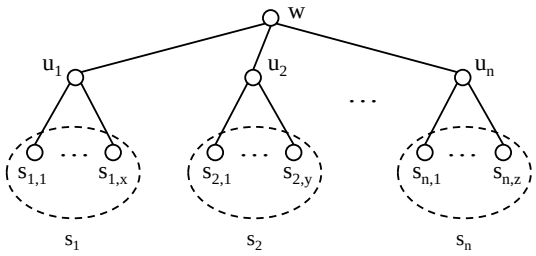


Figure 3: A LOT of height 2 with n super-leaves.

A4 Reliable broadcast functionality is available within a super-leaf, which ensures that all the live nodes in a super-leaf receive the same set of messages.

Proof Sketch: Consider a height-2 LOT consisting of n super-leaves, as shown in Figure 8. Each super-leaf S_i has a height-1 parent vnode u_i. The height-1 vnodes are connected with the common root vnode w. A consensus cycle consists of two rounds for a LOT of height 2. In the first round, the nodes share the pending batch of update-requests within their super-leaf using the reliable broadcast functionality (assumption A4). The nodes sort the messages according to the proposal numbers received in the messages. At the end of the first round, each node $f^i_{S_j}$ in a super-leaf S_j has the same ordered-set of messages M^1_j, which represents the state of the height-1 ancestor u_j of the nodes in S_j. To order the messages in the next round, the largest proposal number received in the current round is selected.

In the second round each node $f^i_{S_j}$ calculates the state of its height-2 ancestor w. Therefore, the set of messages that $f^i_{S_j}$ requires to complete the second round is $M^2_j = \forall_{u \in C^2_j} \cup M^1_u$, where C^2_j is the set of children of the height-2 ancestor of any node in the super-leaf S_j.

By Canopus' design, a set of representatives of S_j fetch the missing state M^1_u of each vnode $u \in C^2_j$ from the live descendants of u. By assumption A1 and A2, the representatives will eventually succeed in fetching the state of u if there is at least one live super-leaf descending from u. If the representatives of S_j eventually succeed in fetching the states of all the vnodes $u \in C^2_j$, then by reliable broadcast within the super-leaf (assumption A4), all the nodes in the S_j have the same set of messages M^2_j that are required to complete the second round.

By assumptions A1 and A3, all the nodes have the same view of LOT in the first consensus cycle. Therefore, the representatives of each super-leaf fetch the same state of a common vnode u from any of the descendants of u. This implies that, for all the super-leaves for which the representatives have succeeded in fetching all the required states of height-1 vnodes, the nodes in the

super-leaves have same set of messages, which allows the nodes to complete the second round. Thus,

$$\underset{i,j \in \widehat{E}_w^2}{\forall} \quad M_i^2 = M_j^2 \tag{2}$$

the nodes sort the messages according the proposal numbers received with the vnode-states. As the consensus cycle consists of two rounds for a LOT of height 2, equation 2 implies that all of the descendants of the height-2 root vnode that complete the first consensus cycle have the same ordered-set of messages. This shows that the live nodes reach agreement and eventually complete the consensus cycle, if a super-leaf does not fail.

If the representatives of a super-leaf S_j fail in fetching the state of at least one vnode $u \in C_j^2$ in the second round, then either (a) all the descendants of u have crashed or they do not have the state of u due to not completing the previous round, or (b) all the representatives of S_j have crashed, which implies that the super-leaf S_j has failed due to insufficient number of live nodes to elect a new representative. In either case, the nodes in S_j cannot complete the second round due to missing a required state, and the consensus process stalls for the nodes in S_j in the second round of the current consensus cycle.

Alternatively, if the nodes in another super-leaf S_k succeed in fetching all the required states, then the consensus process stalls for the nodes in S_k in the next cycle $c+1$ because the live nodes in S_j are stalled in the last cycle c. This shows that the consensus process stalls for the live nodes due to a super-leaf failure and the nodes do not return a wrong result.

In the subsequent consensus cycles, the assumption A1 does not hold because the membership of LOT changes if a node fails or a new node joins a super-leaf. However, we show in the detailed proof, which is given in Appendix A, using induction that the emulation tables are maintained and all the live nodes that eventually complete a cycle c have the same emulation table in cycle $c+1$. Therefore, Canopus satisfies agreement in any consensus cycle for height-2 LOT. We prove these properties for a LOT of any height-h using induction, for which the height-2 LOT serves as the base case.

7 Optimizations

In this section, we discuss two optimizations that enable Canopus to achieve high throughput and low read latency in wide-area deployments.

7.1 Pipelining

The completion time of a consensus cycle depends on the latency between the most widely-separated super-leaves. If only one cycle executes at a time, then high latency would reduce throughput as the nodes are mostly idle and are waiting for distant messages to arrive. Therefore, we use pipelining in Canopus to increase the throughput of the system. Specifically, instead of executing only one consensus cycle at a time, a node is permitted to participate in the next consensus cycle if any of the following events occur:

- The node receives a message belonging to the consensus cycle that is higher than its most recently started consensus cycle.
- A periodical timer expires, which serves as an upper bound for the offset between the start of two consensus cycles.
- The number of outstanding client requests exceeds the maximum batch size of requests.

With this change, a node can start exchanging messages with its peers that belong to the next consensus cycle even before the end of the prior cycle. In effect, this allows Canopus to maintain multiple in-progress consensus cycles. However, log commits only happen when a particular consensus cycle is complete, and always in strict order of consensus cycles. This is no different than a transport layer maintaining a window of transmitted-but-unacked packets.

Two cases need to be addressed to maintain the total order of the requests. First, nodes must always start consensus cycles in sequence (i.e., they cannot skip a cycle). It may happen that a node receives a message belonging to cycle $C_{j \geq i+2}$ where C_i is the most recently started consensus cycle at the node. In that case, the node still starts the next cycle C_{i+1} instead of starting the cycle C_j. Second, nodes always commit the requests from consensus cycles in sequence. Due to parallelism and fetching proposals from different nodes in each cycle, it may happen that a node receives all the required messages to complete a cycle $C_{j \geq i+1}$ before it has completed the cycle C_i. However, to maintain the total order, nodes do not commit the requests from C_j until the requests from all the cycles before C_j have been committed.

7.2 Optimizing Read Operations

Canopus delays read operations to linearize them relative to the write requests across all the nodes in the system. Therefore, the read operations must be deferred to the end of the next consensus cycle, which is bounded by

the round-trip latency between the farthest super-leaves. However, for read-heavy workload, it is desirable that the read operations are performed with minimal latency. Canopus can optionally support write-leases for a particular key during a consensus cycle to reduce the latency of read operations while preserving linearizability.

For any key, during any consensus cycle either a write lease is inactive, so that *no* writes are permitted to that key, and all nodes can read this key with no loss of consistency or *all* nodes have permission to write to this key (with a write order that is decided at the end of the consensus cycle) and no reads are permitted to this key. Any read requests made to a key during a consensus cycle i such that there is a write lease for the key is deferred to the end of the $(i+1)^{th}$ consensus cycle, to ensure linearizability.

Write leases require the following three modifications to Canopus:

1. Blocking client requests: Following the model in Paxos Quorum Leases [31], our read optimization restricts the clients to sending only *one* request at any one time. This is different than our earlier model, where the clients are permitted many outstanding requests at a time.

2. Write leases: Write requests are committed after acquiring the write-lease for the keys in the requests. Nodes piggyback lease requests with the proposal messages in the next consensus cycle C_{i+1}. At the end of the consensus cycle C_{i+1}, all the correct nodes that complete the cycle have the same set of lease requests. Therefore, all the correct nodes in the system can apply the lease in the same consensus cycle C_{i+p+1}.

3. Reads without delay: A node N performs a read operation for a key y immediately (reading results from committed consensus cycles) if the write-lease for y is not active in any of the currently ongoing consensus cycles. If a node receives a read request for y while the write-lease for y is currently active, the node delays the read request until the end of the next consensus cycle.

8 Evaluation

The focus of our evaluation is to compare the throughput and latency of Canopus with other competing systems at different deployment sizes. Our experiments are based on our prototype implementation of Canopus, which consists of approximately 2000 lines of Java code. We select EPaxos as a representative for state-of-the-art decentralized consensus approaches as EPaxos has been reported [28] to perform equal or better than other decentralized consensus protocols. We use the publicly available implementation [29] of EPaxos from the paper's authors for the evaluation. We also compare against ZooKeeper, a widely used coordination service that uses Zab, a centralized atomic broadcast protocol, to provide agreement. In order to provide a fair comparison with ZooKeeper, which includes additional abstractions that may introduce extra overhead, we created ZKCanopus, a modified version of ZooKeeper that replaces Zab with Canopus and performed all of our comparisons to ZooKeeper using ZKCanopus.

8.1 Single Datacenter Deployment

In this section, we evaluate the performance of Canopus and EPaxos when all the nodes are located in the same datacenter.

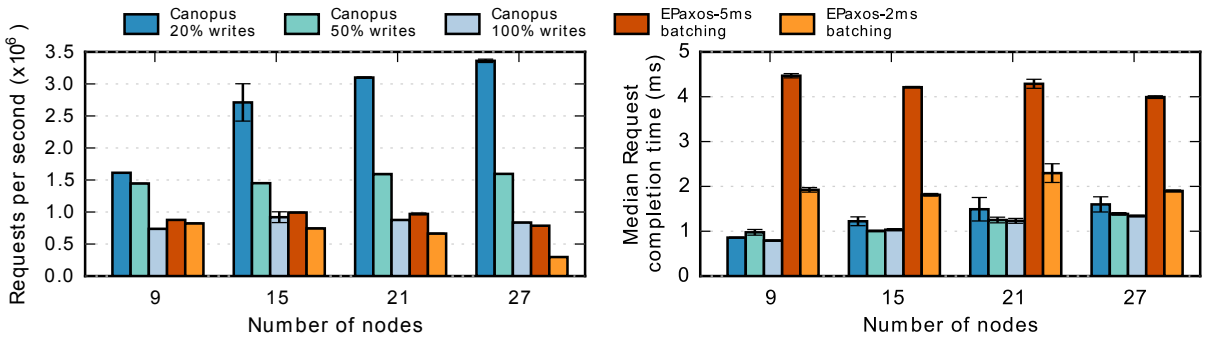
Experimental setup: We conducted our experiments on a three rack cluster where each rack consists of 13 machines. Each machine contains 32GB of memory, a 200GB Intel S3700 SSD, and two Intel Xeon E5-2620 processors. The machines in each rack are connected through a Mellanox SX1012 top-of-rack switch via 10Gbps links. Rack switches are connected through 2x10Gbps links to a common Mellanox SX1012 aggregation switch. Thus, the network oversubscription is 1.5, 2.5, 3.5, and 4.5 for experiments with 9, 15, 21, and 27 nodes respectively.

For our ZooKeeper and ZKCanopus experiments, both systems were configured to write logs and snapshots of the current system state asynchronously to the filesystem. We use an in-memory filesystem to simplify our experimental setup. To ensure that using an in-memory filesystem does not appreciably affect our results, we perform additional experiments in which logs are stored to an SSD. The results show that, for both ZooKeeper and ZKCanopus, throughput is not affected and median completion time increases by less than 0.5 ms.

Each experiment runs for 50 seconds and is repeated five times. We discard the first and last 5 seconds to capture the steady state performance. The error bars in the graphs show 95% confidence intervals.

Workload: Our experimental workloads are driven by 180 clients, which are uniformly distributed across 15 dedicated machines with 5 machines per rack. Each client connects to a uniformly-selected node in the same rack. The clients send requests to nodes according to a Poisson process at a given inter-arrival rate. Each request consists of a 16-byte key-value pair where the key is randomly selected from 1 million keys.

LOT configuration in Canopus: Canopus nodes are organized into three super-leaves in a LOT of height 2. To vary the group size, we change the super-leaf size to 3, 5, 7, and 9 nodes. Due to the size of our cluster, we did not run experiments with taller trees. In our current im-



(a) Throughput

(b) Median completion time

Figure 4: Throughput and median request completion times of Canopus and EPaxos while scaling the number of nodes in the system.

plementation, nodes reliably broadcast messages within a super-leaf without hardware assistance. Instead, they use the Raft-based reliable broadcast protocol described in Section 4.3 in which nodes communicate using unicast messages.

Performance metrics: The two key performance metrics we measure in our experiments are throughput and request completion time. We determine the throughput of a system by increasing the request inter-arrival rate until the throughput reaches a plateau. Our empirical results show that, for all of the tested systems and configurations, the plateau is reached before the median request completion time reaches 10 ms. To simplify our testing procedure, our experiments run until the request completion time is above 10 ms and we use the last data point as the throughput result for that run. For each run, we manually verify that we have reached the system's maximum throughput.

As it is difficult to determine the exact inter-arrival rate at which maximum throughput has been reached, together with the common operating practice of running critical systems below their maximum load, we report the median request completion time of the tested systems when they are operating at 70% of their maximum throughput. We believe this is more representative of the completion times that applications will experience when interacting with these systems compared to the median completion time at 100% load.

8.1.1 Comparison with EPaxos

In this experiment, we compare the performance of Canopus with EPaxos at different system sizes. Figure 4(a) shows the maximum throughput of the two systems. The first three bars in each group show the Canopus' throughput at 20%, 50%, and 100% write requests, and the last two bars show the throughput of EPaxos at its default batch duration of 5 ms, in which requests are delayed for up to 5 ms in order to collect a larger batch of requests, and a reduced batch duration of 2 ms. For EPaxos, we ensure there is 0% command interference to evaluate its performance under the best possible condition. As EPaxos sends reads over the network to other nodes, its performance is largely independent of the write percentage; we show its results using the 20% write request workload.

The performance results show that with 20% and 50% write requests, Canopus provides significantly higher throughput than EPaxos with the performance gap increasing with larger system sizes. With 27 nodes and 20% write requests, Canopus provides more than 3 times the throughput of EPaxos with 5 ms batching. This is because Canopus can serve read requests locally without disseminating the request to the rest of the nodes in the group while still providing linearizable consistency. For the 100% write request workload, Canopus provides similar throughput to EPaxos with a 5 ms batching duration. This is in part due to network bandwidth being relatively abundant in a single datacenter deployment. EPaxos with a 2 ms batching duration shows a significant drop in throughput as we increase the number of nodes in the system. This illustrates that EPaxos has scalability limitations when configured with smaller batch sizes.

Figure 4(b) shows the median request completion times of the two systems. From 9 to 27 nodes, Canopus' request completion time is mostly independent of write request percentage, and is significantly shorter than EPaxos.

EPaxos' high request completion time is primarily due to its batching duration. By reducing its batching duration to 2 ms, EPaxos' median request completion time reduces by more than half and is within 1 ms of Canopus' request completion time. However, as

we saw previously, EPaxos relies on large batch sizes to scale. Therefore, our results show that EPaxos must tradeoff request completion time for scalability, whereas Canopus' read throughput increases with more nodes, its write throughput remains largely constant up to at least 27 nodes, and its median request completion time only marginally increases going from 9 to 27 nodes.

8.1.2 Comparison with ZooKeeper

In this experiment, we compare the performance of ZooKeeper with ZKCanopus, our Canopus-integrated version of ZooKeeper. This comparison serves three purposes. First, it compares the performance of Canopus with that of a centralized coordinator-based consensus protocol. Second, it shows the effectiveness of Canopus in scaling an existing system by eliminating the centralized coordinator bottleneck. Finally, these experiments evaluate the end-to-end performance of complete coordination services, instead of evaluating just the performance of the consensus protocols.

We configure ZooKeeper to have only five *followers* with the rest of the nodes set as *observers* that do not participate in the Zab atomic broadcast protocol but asynchronously receive update requests. This choice is mainly to reduce the load on the centralized *leader* node. Although observers do not provide additional fault tolerance, they can improve read request performance by servicing read requests. In the case of ZKCanopus, every node participates fully in the Canopus consensus protocol. We use one *znode* in these experiments and the clients read from and write to the same znode.

Figure 5 shows the throughput to median request completion time results for ZKCanopus and ZooKeeper with 9 and 27 nodes. For ZKCanopus, we do not show the request completion time at low throughputs in order to improve the readability of the graphs. The results show that, when the system is not loaded, ZKCanopus has a median request completion time that is between 0.2 ms to 0.5 ms higher than ZooKeeper. This is in part due to the additional network round trips required to disseminate write requests using a tree overlay instead of through direct broadcast. However, we believe the small increase in request completion time is acceptable given the significant improvement in scalability and throughput that ZKCanopus provides over ZooKeeper.

8.2 Multi-Datacenter Deployment

In this section, we evaluate the performance of Canopus in a wide area deployment. We run the experiments on Amazon EC2 cloud using three, five, and seven datacenters. Each datacenter consists of three nodes located in the same site. Each node runs on an EC2 c3.4xlarge instance, which consists of 30GB memory and Intel Xeon E5-2680 processor. Each datacenter has 100 clients that connect to a uniform-randomly selected node in the same datacenter and concurrently perform read and write operations at a given rate. The workload consists of 20% write requests, unless otherwise mentioned. The messages consists of 16-byte key-value pairs. The inter-datacenter latencies are given in the Table 1.

Table 1: Latencies (ms) between the datacenters used in the experiments.

	IR	CA	VA	TK	OR	SY	FF
IR	0.2						
CA	133	0.2					
VA	66	60	0.25				
TK	243	113	145	0.13			
OR	154	20	80	100	0.26		
SY	295	168	226	103	161	0.2	
FF	22	145	89	226	156	322	0.23

For Canopus, each datacenter contains a super-leaf, although the nodes might not be located in the same rack. We enable pipelining to overcome the long delays across datacenters. Each node starts a new consensus cycle every 5ms or after 1000 requests have accumulated, whichever happens earlier. For EPaxos, we used the same batch size as Canopus, and the workload consists of zero command interference. We enable latency-probing in EPaxos to dynamically select the nodes in the quorum. We disable thrifty optimization in EPaxos because our experiments show lower throughput with the thrifty optimization turned on.

Figure 6 shows the throughput and median request completion times of Canopus and EPaxos when deployed in three, five, and seven datacenters. Each datacenter consists of three nodes. The vertical lines in the graph show the throughput when the latency touches 1.5 times the base latency.

Scaling across the datacenters, Canopus is able to achieve about 2.6, 3.8, and 4.7 millions requests per second, which is nearly 4x to 13.6x higher throughput than EPaxos. Canopus is able to achieve high throughput because of its more efficient utilization of CPU and network resources. Unlike EPaxos, Canopus does not disseminate read requests across the nodes, not even the read keys, which reduces the number and size of messages exchanged between the nodes. For ready-heavy workloads, this significantly impacts the utilization of CPU and network resources. Furthermore, due to the hierarchical and network-aware design of Canopus, a proposal message is exchanged between each pair of super-leaves only once. This approach places a lower load

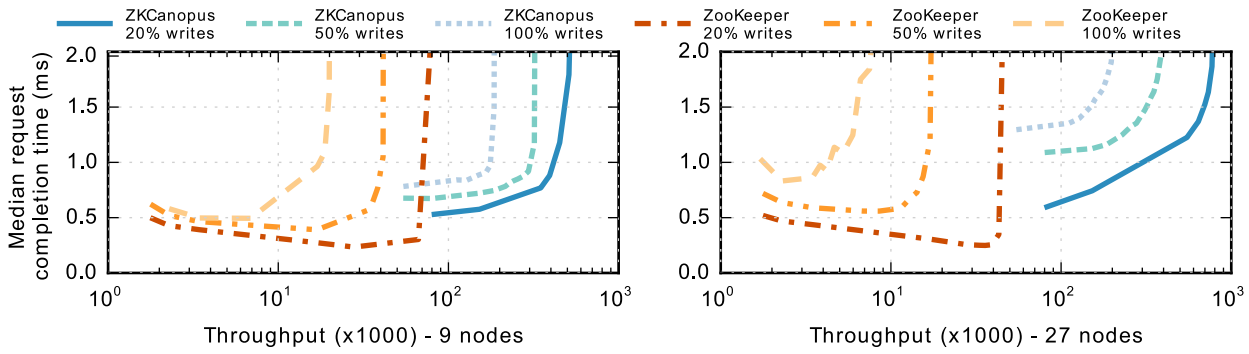


Figure 5: Throughput and median request completion times of ZKCanopus and ZooKeeper when the system consists of 9 nodes (left) and 27 nodes (right). Note that the x-axis is in log scale.

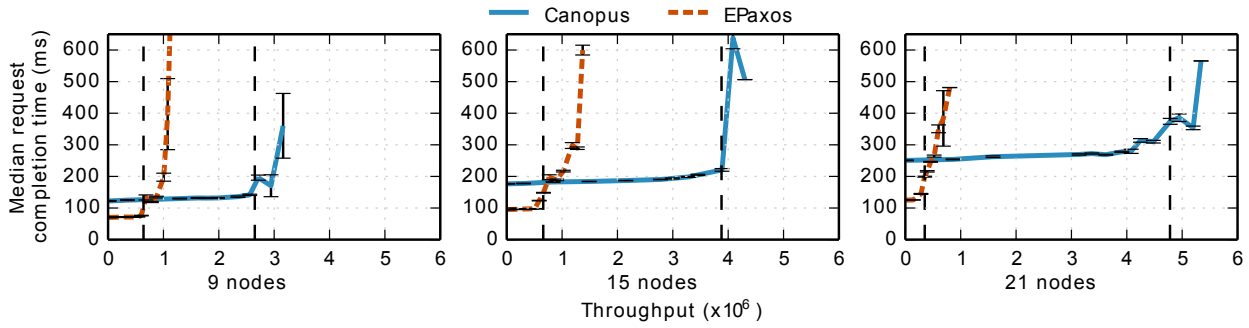


Figure 6: Throughput and median request completion times of Canopus and EPaxos in the multi-datacenter experiment with 20% writes workload.

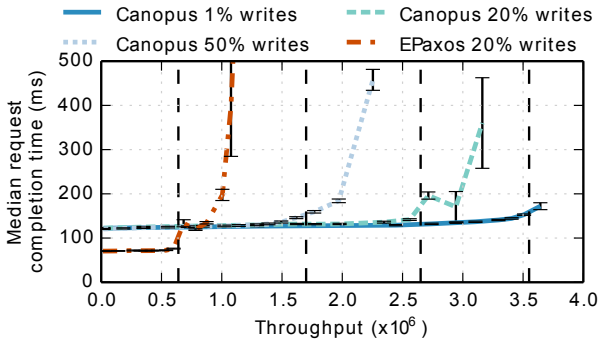


Figure 7: Performance of Canopus and EPaxos with different ratios of writes in the workload.

on wide-area network links compared to the broadcast-based approach used in EPaxos.

8.2.1 Writes to Reads Ratio in the Workload

In this section, we evaluate the performance of Canopus and EPaxos with different ratios of writes in the workload. We run these experiments with nine nodes deployed in three datacenters. Figure 7 shows the results when the workload consists of 1%, 20%, and 50% write requests. For EPaxos, our results show the same results with different write-ratios in the workloads. Therefore, we only show the results for 20% writes workload.

As expected, Canopus has higher throughput for more read-heavy workloads, reaching up to 3.6 million requests per second with 1% writes workload , as compared to the 2.65 millions requests per second with 20% writes workload. However, even with a write intensive workload consisting of 50% writes, Canopus achieves 2.5x higher throughput than EPaxos.

9 Discussion

This paper describes the current design of Canopus and evaluates its performance in comparison with other protocols. Additional capabilities may be necessary in order to satisfy the requirements of specific applications.

Full-rack failures: In Canopus, nodes belonging to the same super-leaf are placed in the same rack. Therefore, catastrophic failures that cause complete rack failures result in super-leaf failures, which halts the consensus process. Although, top-of-rack switch failures are rare [13] and the fault-tolerant design of datacen-

ters [22, 23, 39, 37] reduce the possibility of rack failures, there can be applications that have more stringent availability requirements. For such applications, we are currently investigating a rack-failure recovery mechanism.

Experiments at large scale: In order to limit the cost of our experiments, we only evaluated Canopus with up to 27 nodes across seven datacenters. However, we expect Canopus to scale to a large number of nodes by carefully constructing the LOT hierarchy. The number of nodes in Canopus can be increased either by increasing the size of the super-leaves, or by increasing the number of super-leaves in the system. The LOT should be structured such that the time it takes to complete the operations within a super-leaf is less than the round-trip time between super-leaves. If necessary, the height of the tree can be increased to satisfy this condition.

Byzantine fault tolerance: Private blockchains use consensus protocols to achieve agreement on the ordering and timing of operations between the participating nodes. In many private blockchains, nodes do not trust each other as they can act maliciously, either intentionally or due to being hacked. A common approach to solve the trust problem is to use Byzantine fault-tolerant (BFT) consensus algorithms. An alternative approach to establish trust is to create a trusted computing environment [4] using hardware assisted secure enclaves. In this way, the nodes can use non-BFT consensus protocols, such as Canopus, to solve the trust issue. We are in the process of extending Canopus to support BFT.

10 Conclusion

In this paper, we have presented Canopus, a highly parallel, scalable, and network-aware consensus protocol. Evaluation results using our prototype implementation show that Canopus can perform millions of requests per second with 27 nodes in a single datacenter deployment, which is more than 4x higher than EPaxos. Furthermore, our Canopus-integrated version of ZooKeeper increases the throughput of ZooKeeper by more than 16x for read-heavy workloads.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). This work benefited from the use of the SyN Facility at the University of Waterloo.

References

[1] Chain blockchain infrastructure, 2017.

[2] Hyperledger blockchain technologies, 2017.

[3] Kadena permissioned blockchain, 2017.

[4] The Coco Framework, 2017.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74. ACM, 2008.

[6] A. Allavena, Q. Wang, I. Ilyas, and S. Keshav. LOT: A robust overlay for distributed range query processing. Technical report, CS-2006-21, University of Waterloo, 2006.

[7] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[8] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-Paxos: Offloading the leader for high throughput state machine replication. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 111–120. IEEE, 2012.

[9] H. T. Dang, M. Canini, F. Pedone, and R. Soulé. Paxos made switch-y. *SIGCOMM Computer Communication Review (CCR)*, 46(2):18–24, May 2016.

[10] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*, pages 5:1–5:7. ACM, 2015.

[11] N. Farrington and A. Andreyev. Facebook's data center network architecture. In *IEEE Optical Interconnects Conference*, pages 5–7, 2013.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[13] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM '11, pages 350–361. ACM, 2011.

[14] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. *Communications of the ACM*, 54(3):95–104, 2011.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC '10, page 9. USENIX Association, 2010.

[16] Z. István, D. Sidler, G. Alonso, and M. Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '16, pages 425–438, Santa Clara, CA, 2016. USENIX Association.

[17] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN '11)*, pages 245–256. IEEE, 2011.

[18] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[19] L. Lamport. Generalized consensus and paxos. Technical report, MSR-TR-2005-33, Microsoft Research, 2005.

[20] L. Lamport. Fast paxos. *Springer Distributed Computing*, 19(2):79–103, 2006.

[21] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 467–483. USENIX Association, 2016.

[22] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *In Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 399–412. USENIX Association, 2013.

[23] V. Liu, D. Zhuo, S. Peter, A. Krishnamurthy, and T. Anderson. Subways: A case for redundant, inexpensive data center edge links. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 27:1–27:13. ACM, 2015.

[24] R. Makhloufi, G. Doyen, G. Bonnet, and D. Gaïti. A survey and performance evaluation of decentralized aggregation schemes for autonomic management. *International Journal of Network Management*, 24(6):469–498, 2014.

[25] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8 of *OSDI '08*, pages 369–384, 2008.

[26] P. J. Marandi, M. Primi, and F. Pedone. Multi-ring paxos. In *Proceedings of the 2012 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '12, pages 1–12. IEEE, 2012.

[27] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems Networks*, DSN '10, pages 527–536. IEEE, 2010.

[28] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372. ACM, 2013.

[29] I. Moraru, D. G. Andersen, and M. Kaminsky. EPaxos source code, 2014.

[30] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of SoCC*. ACM, 2014.

[31] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–13. ACM, 2014.

[32] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of USENIX OSDI*, pages 517–532. USENIX Association, 2016.

[33] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of USENIX ATC*, pages 305–320. USENIX Association, 2014.

[34] M. Poke, T. Hoefler, and C. W. Glass. Allconcur: Leaderless concurrent atomic broadcast. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, HPDC '17, pages 205–218. ACM, 2017.

[35] D. R. K. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 43–57, Oakland, CA, 2015. USENIX Association.

[36] J. Risson and T. Moors. Survey of research towards robust peer-to-peer networks: search methods. *Computer networks*, 50(17):3485–3521, 2006.

[37] B. Schlinker, R. N. Mysore, S. Smith, J. C. Mogul, A. Vahdat, M. Yu, E. Katz-Bassett, and M. Rubin. Condor: Better topologies through declarative design. In *Proceedings of the ACM SIGCOMM 2015 Conference*, pages 449–463. ACM, 2015.

[38] X. Shi, H. Lin, H. Jin, B. B. Zhou, Z. Yin, S. Di, and S. Wu. GIRAFFE: A scalable distributed coordination service for large-scale systems. In *Proceedings of IEEE Cluster Computing (CLUSTER)*, pages 38–47. IEEE, 2014.

[39] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Porceedings of SIGCOMM*, pages 183–197. ACM, 2015.

[40] P. Sutra and M. Shapiro. Fast genuine generalized consensus. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 255–264. IEEE, 2011.

[41] R. Van Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems (TOCS)*, 21(2):164–206, May 2003.

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

Symbol	Description
h	Tree height
$S = \{ s_i \}$	Set of super-leaves with elements s_i
$s_{i,j}$	j^{th} leaf-node in a super-leaf s_i
$L = \{ l_{i,j} \}$	Set of correct leaf nodes where $l_{i,j} \in s_i$
$A(n, i)$	i^{th} ancestor of (v)node n; $a(n, 0) = n$
$C(n)$	Set of children of node n
$D(n)$	Correct descendants of node n
$R(s_{i,j}, c)$	Set of requests that a node $s_{i,j}$ has received during consensus cycle $c - 1$.
$F(s_{i,j}, c)$	Set of membership updates that a node $s_{i,j}$ has received during consensus cycle $c - 1$. Membership updates are of the form $delete(s_{i,j})$ and $add(s_{i,j})$ which delete and add a leaf node in super-leaf s_i respectively.
$N(s_{i,j}, c)$	A large random number (e.g., 32-bit) chosen by node $s_{i,j}$ at the start of consensus cycle c
$\Pi(s_{i,j}, n, c, r)$	State of (v)node n computed at leaf node $s_{i,j}$ just before the *start* of the r^{th} round of consensus cycle c. The state of a leaf node (at itself) consists of the nested tuple $< F(s_{i,j}, c), < N(s_{i,j}, c), R(s_{i,j}, c) >>$. The state of a non-leaf node n at leaf node $s_{i,j}$ consists of the nested tuple $< F(c), [< N(s_{i',j'}, c), R(s_{i',j'}, c) >] >$ where $F(c)$ is the set of membership updates received by node $s_{i,j}$ during cycle c, and $[< N(s_{i',j'}, c), R(s_{i',j'}, c) >]$ is an array of requests received by $s_{i,j}$ from leaf nodes $s_{i',j'} \in D(n)$ ordered by $N(s_{i',j'}, c)$.
$\Pi^t(s_{i,j}, n, c, r)$	Temporary state of vnode n at leaf node $s_{i,j}$ during the r^{th} round of consensus cycle c. This state consists of the tuple $< F^t(c), [< N(s_{i',j'}, c), R(s_{i',j'}, c) >]^t >$, where $F^t(c)$ is the set of membership updates received so far in round r by node $s_{i,j}$, and $[< N(s_{i',j'}, c), R(s_{i',j'}, c) >]^t$ is the set of requests received by $s_{i,j}$ from leaf nodes $s_{i',j'} \in D(n)$ ordered by $N(s_{i',j'}, c)$.
$M(s_{i,j}, \Pi(s_{i,j}, n, c, r), c, r)$	Proposal response that a node $s_{i,j}$ in super-leaf s_i sends in round r of consensus cycle c
$T(s_{i,j}, n, c)$	Emulation table at node $s_{i,j}$ for node n at the start of consensus cycle c. This is a list of leaf nodes that emulate n.

Table 2: Notations used in the appendix

A Correctness Proof

In this section, we provide a proof of correctness for Canopus. This proof has minor differences in notation and structure to our proof-sketch in Section 6.

A.1 Definitions

We first define the terms used in the proof.

1. **Leaf-nodes:** A *leaf-node* in LOT is a physical server implementing the Canopus protocol. The terms *leaf-node* and *node* are used interchangeably. Each node is assumed to have a globally unique *node ID*.

2. **Virtual-nodes:** A *virtual-node*, also denoted as *vnode*, is any non-leaf node. Each vnode is also assumed to have a globally unique *node ID*.

3. **Super-leaf:** Nodes are logically grouped to form *super-leafs*. All nodes in a super-leaf share the same height-1 parent vnode, that is, $\forall j, j'$ in the same super-leaf s_i $A(s_{i,j}, 1) = A(s_{i,j'}, 1)$.

4. **Reliable broadcast:** A reliable broadcast[2] ensures that broadcast communication within a super-leaf has *validity*, *integrity*, and *agreement* (these are defined further in the cited reference).

5. **Consensus cycle and round:** A *consensus cycle* is a complete execution of the Canopus consensus protocol. The consensus cycle consists of h rounds, where h is the height of the root node. The state of the root node represents the consensus, and this is computed at the end of h rounds. We use the terms *consensus cycle* and *cycle* interchangeably.

6. **Super-leaf representative:** Each super-leaf s_i elects k nodes as *representatives* of that super-leaf. A super-leaf representative fetches remote state on behalf of its peer nodes in its super-leaf, then uses reliable broadcast to disseminate the fetched state to live nodes in its super-leaf.

7. **Failed node:** A failed node is a crashed node. A node that has not failed is called **alive**. A failed node is unable to communicate with live nodes.

   - Each node in a super-leaf is the leader of its own Raft group. Therefore, upon failure of a node its group will know the leader has failed and will elect a new leader. This failure detector is assumed to be perfect, and its output is used to update the membership state at each leaf node, then subsequently communicated to other nodes using message exchanges. Failed nodes recover by explicitly rejoining the system using a join protocol, which also updates the membership state at nodes.
   - If a super-leaf representative fails, the remaining live nodes in the super-leaf elect a new representative to replace the failed one. A new representative can be elected using a leader election protocol. In our implementation, we use the Raft consensus protocol within a super-leaf, which provides this functionality.
   - Note that the successful fetching of remote state by *any one* of the k representatives allows the Canopus protocol to proceed despite crash-stop failures. Thus, a sufficiently large value of k mitigates both against representative failure as well as long network latencies in the state fetching process.

8. **Suspended node:** A suspended node is alive but cannot progress. A node eventually goes into the suspended state if either (a) its super-leaf has failed due to an insufficient number of live nodes, or (b) the node is missing a required message to complete the current round in a consensus cycle. A suspended node can receive messages and can share state or messages belonging to previous rounds that it has already completed. A suspended node becomes correct and can progress when it has received all the required messages to complete the current round.

9. **Correct node:** A node is correct if it is both alive and can make progress, i.e., it is not suspended. Only correct nodes start and complete a consensus cycle.

10. **Super-leaf failure:** A super-leaf s_i is considered to have failed if the live nodes in s_i do not have sufficient quorum to elect k representative for s_i. This can happen if the election process require more than b live nodes and the super-leaf does not have more than b live nodes. If a super-leaf fails, the live nodes in the super-leaf go into a suspended state.

11. **Message:** All correct leaf nodes calculate the state of their height-r ancestors in the round r of each consensus cycle (as discussed later in this section). Upon request from peers or super-leaf representatives, they send this calculated state using proposal-response messages. For simplicity, we only model proposal responses and not proposal requests. We assume messages are either transferred in their entirety from source to destination and are received without *duplication* or *corruption* or lost.

    A proposal response message $M(s_{i,j}, \Pi(s_{i,j}, n, c, r-1), c, r)$ sent by node $s_{i,j}$ during round r contains the state $\Pi(s_{i,j}, A(s_{i,j}, r-1), c, r)$ of n, the height-$(r-1)$ ancestor $A(s_{i,j}, (r-1))$ of $s_{i,j}$. For example, in the first round, when $r = 1$, since $A(s_{i,j}, 0) = s_{i,j}$, $s_{i,j}$ sends it own state computed just before the start of round 1 in each message $M(s_{i,j}, 1)$. We discuss how this state is computed next.

12. **Node state**: The state of node/vnode n at node $s_{i,j}$ just before start of round r of consensus cycle c, denoted $\Pi(s_{i,j}, n, c, r)$. Node state is computed recursively, as discussed next.

    To begin with, the state of a leaf node $s_{i,j}$ at itself just before the start of consensus cycle c, denoted $\Pi(s_{i,j}, s_{i,j}, c, 1)$ (i.e., just before round 1) is defined as

$$\Pi(s_{i,j}, s_{i,j}, c, 1) \triangleq < F(s_{i,j}, c), < N(s_{i,j}, c), R(s_{i,j} >, c) > \tag{3}$$

[2]Hadzilacos V, Toueg S. A modular approach to fault-tolerant broadcasts and related problems. Cornell University; 1994.

where

- $F(s_{i,j}, c)$ is the set of membership updates that $s_{i,j}$ has received, but not communicated, before starting consensus cycle c. This is the set of nodes that have been detected to have failed or have joined before starting the current cycle c.
- $N(s_{i,j}, c)$ is a long (e.g. 32-bit) random number. Each leaf node $s_{i,j}$ randomly generates $N^c_{i,j}$ before the start of the first round of consensus cycle c.
- $R(s_{i,j}, c)$ is the set of requests that $s_{i,j}$ has received, but not communicated, before starting consensus cycle c. In other words, these requests are not part of $\Pi(s_{i,j}, A(s_{i,j}, h), (c-1), (h+1))$. In other words, these changes are not reflected in $\Pi(s_{i,j}, A(s_{i,j}, h), (c-1), (h+1))$.

The state of a non-leaf node n at leaf node $s_{i,j}$ consists of the nested tuple $< F(c), [< N(s_{i',j'}, c), R(s_{i',j'}, c) >] >$ where $F(c)$ is the set of membership updates received by node $s_{i,j}$ during cycle c, and $[< N(s_{i',j'}, c), R(s_{i',j'}, c) >]$ is the set of requests received by $s_{i,j}$ from leaf nodes $s_{i',j'} \in D(n)$ ordered by $N(s_{i',j'}, c)$. It is computed as follows.

Immediately after the start of the r^{th} round of consensus cycle c, node $s_{i,j}$ initializes its temporary copy of its state for its r^{th} level ancestor $A(s_{i,j}, r)$ as its state for its $(r-1)^{th}$ ancestor $A(s_{i,j}, r-1)$:

$$\Pi^t(s_{i,j}, A(s_{i,j}, r), c, r) =< F^t(c), [< N(., c), R(., c) >]^t > \leftarrow \Pi(s_{i,j}, A(s_{i,j}, r-1), c, r) \tag{4}$$

Let node $s_{i,j}$ receive $M(s_{i',j'}, \Pi(s_{i',j'}, A(s_{i',j'}, r-1), c, r), c, r)$ either in its role as a super-leaf representative, or as part of a reliable broadcast from its representative. This message originates from node $s_{i',j'}$ in round r of consensus cycle c and carries that node's state for its $(r-1)^{th}$ ancestor $A(s_{i',j'}, r-1)$. The node first checks if this is a duplicate message, that it, it already has received the state of $A(s_{i',j'}, r-1)$. If not, $s_{i,j}$ updates its temporary state $\Pi^t(s_{i,j}, A(s_{i,j}, r), c, r)$ as follows:

$$\Pi^t(s_{i,j}, A(s_{i,j}, r), c, r) \leftarrow \Pi^t(s_{i,j}, A(s_{i,j}, r), c, r) \sqcup \Pi(s_{i',j'}, A(s_{i',j'}, r-1), c, r) \tag{5}$$

where the union operation $\sqcup$ is defined by:

$$F^t(c) \leftarrow F^t(c) \cup F(s_{i',j'}, c) \tag{6}$$

$$[< N(., c), R(., c) >]^t \leftarrow [< N(., c), R(., c) >]^t || < N(s_{i',j'}, c), R(s_{i',j'}, c) > \tag{7}$$

where $||$ indicates addition of a tuple into the correct location in an array of ordered tuples.

Due to the use of reliable broadcast, either all live nodes in a super-leaf receive this message or none do. Hence, all nodes in the super-leaf consistently update their temporary state after each message broadcast. Finally, node $s_{i,j}$ updates the state of its r^{th} ancestor $A(s_{i,j}, r)$ just before the start of the next round $r+1$ to the temporary state:

$$\Pi(s_{i,j}, A(s_{i,j}, r), c, r+1) = \Pi^t(s_{i,j}, A(s_{i,j}, r), c, r) \tag{8}$$

Note that the state $\Pi(s_{i,j}, A(s_{i,j}, r), c, r+1)$ of n, a height-r ancestor of node $s_{i,j}$ at the end of round r of consensus cycle c is defined the union of the states of its children $C(A(s_{i,j}, r))$ at the start of round r. Hence, we can summarize state computation as follows:

$$\Pi(s_{i,j}, s_{i,j}, c, 1) \triangleq < F(s_{i,j}, c), < N(s_{i,j}, c), R(s_{i,j}, c) >> \tag{9}$$

$$\Pi(s_{i,j}, A(s_{i,j}, r), c, r+1) = \bigsqcup_{\forall\, n' \in C(A(s_{i,j}, r))} \Pi(s_{i,j}, n', c, r), \quad 1 \le r \le h \tag{10}$$

$$\Pi(s_{i,j}, A(s_{i,j}, k), c, r+1) = \Pi(s_{i,j}, A(s_{i,j}, k), c, r), \quad 1 \le k < r \le h \tag{11}$$

$\Pi(A(s_{i,j}, k), c, r+1)$ is undefined otherwise.

The first equation is the base condition that initializes the recursion. The second equation captures the parallel computation in each round of computation. The third equation reflects the fact that once a vnode's state is computed, it does not change for the remainder of the consensus cycle.

13. **End of a consensus cycle:** Round r of consensus cycle c at node $s_{i,j}$ ends when it is able to compute $\Pi(s_{i,j}, A(s_{i,j}, r), c, r+1)$. That is, it has received a message carrying the state of every child $C(A(s_{i,j}, r))$ of its ancestor $A(s_{i,j}, r)$. Consensus cycle c ends at node s_{ij} when it completes the h^{th} round of cycle c. Note that different nodes may complete the same consensus cycle at different times.

14. **Emulation table:** An emulation table $T(s_{i,j}, n, c)$ at node $s_{i,j}$ at the start of consensus cycle c maps a vnode n to the list of its emulators, i.e., its correct descendants. A super-leaf representative queries its emulation table to select an emulator of a vnode n from which the representative wants to fetch the state of n. The emulation table contains the necessary information about the emulators of each vnode such as their IDs and the IP addresses. During initialization, the emulation table maps every vnode to *all* of its descendants.

15. **Update and dissemination of the emulation table**: As nodes fail, join, and re-join, the emulation table is updated. Specifically, if the failure detection protocol within a super-leaf indicates that a certain node $s_{i,j}$ has failed during consensus cycle c, then the remaining nodes in s_i update their emulation table to remove $s_{i,j}$ as an emulator for every ancestor vnode and also remove the state from this node from $\Pi^t(s_{i,j}, A(s_{i,j}, 1), c, r)$, their local temporary state for their shared parent vnode. Moreover, the identity of this node is added to $F(s_{i,j}, c)$ and shared with all other nodes in the next consensus cycle $c+1$.

    A node $s_{i,j}$ joins or re-joins super-leaf s_i during consensus cycle c by sending a reliable broadcast to all other nodes in s_i. If successful, this results in three outcomes:

    - The fact that $s_{i,j}$ is now part of super-leaf s_i is indicated in the membership update $F(s_{i,j}, c+1)$ that is part of the consensus and hence carried in all proposal response messages originating from other nodes $s_{i,j'} \in s_i$ in consensus cycle $c+1$.
    - At the end of cycle $c+1$, all live nodes (if any) who receive this broadcast add the newly joined node $s_{i,j}$ to the list of emulators for all ancestors of the joining node in the emulation table
    - Starting with cycle $c+2$, this node's state is required for computing the state for its parent vnode s_i when ending the first round of each cycle.

    Until the end of cycle $c+1$, this node is not considered to be live.

A.2 Assumptions

The proof is based on the following assumptions:

A1 All nodes are initialized with the same emulation table and membership view.

A2 The network does not partition. Moreover, within a super-leaf, all messages are always delivered to a live receiver within a bounded time and therefore node failure can be perfectly detected. Nodes fail by crashing and are not Byzantine.

A3 Super-leaf-level structure does not change: Nodes may leave or join super-leaves, however, super-leafs are neither added nor removed.

A4 Super-leaves support reliable broadcast functionality, guaranteeing*validity*, *integrity*, and *agreement*. In our implementation, we used Raft to provide reliable broadcast. Moreover, messages are delivered without duplication or corruption.

A.3 Proof

We want to prove the consensus property that all correct nodes that complete a consensus cycle c commit the same ordered-set of messages at the end of the cycle c. We will prove a slightly stronger claim:

THEOREM 1: In a height-h LOT consisting of n super-leaves, at the end of the h^{th} round of a consensus cycle, all live nodes that complete the last (h^{th}) round are either stalled or have the same state for the root node:

$$\underset{l_{i,j}, l_{i',j'} \in L}{\forall} \Pi(l_{i,j}, A(l_{i,j}, h), c, h+1) = \Pi(l_{i',j'}, A(l_{i',j'}, h), c, h+1) \tag{12}$$

We prove Theorem 1 by induction, as sketched next:

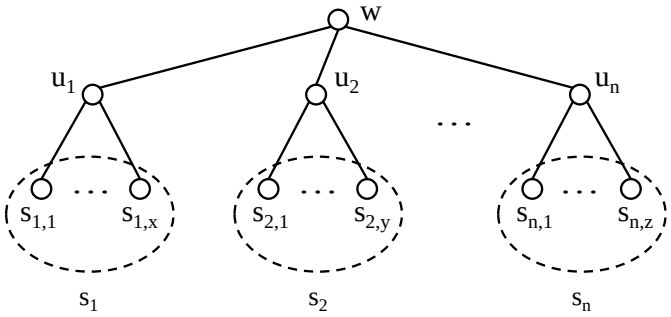


Figure 8: LOT of height 2 with n super-leaves

- We first show that the theorem holds for the first cycle of a tree of height 2 (Lemma 1).
- We then show that, assuming that the theorem holds for a tree of height k, the theorem is true at the end of the last round $r = k + 1$ of the first cycle for a tree of height $k + 1$ (Lemma 2).
- We finally show that, assuming that the theorem holds at the end of the $(c-1)^{th}$ cycle, it also holds at the end of the c^{th} cycle (Lemma 3).

A.4 First cycle for tree of height 2

Consider a tree of height 2, consisting of n super-leaves, $S = \{s_1, s_2, \ldots, s_n\}$, as shown in Figure 8. Super-leaves s_i correspond to vnodes u_i, with a common parent height-2 vnode w. The consensus cycle consists of two rounds, r_1 and r_2. In the first round, states of vnodes u_i are calculated, and in the second round, the state of the root-vnode w is calculated.

LEMMA 1 For a tree of height 2, at the end of the first consensus cycle, all the correct nodes descending from the height-2 root-vnode w, that complete the second round have the same state for the root node:

$$\underset{l_{i,j}, l_{i',j'} \in C(C(w))}{\forall} \Pi(l_{i,j}, w, 1, 3) = \Pi(l_{i',j'}, w, 1, 3) \tag{13}$$

Moreover, the consensus process is stalled for all nodes that do not complete the second round of the first consensus cycle.

PROOF: The first cycle has two rounds.

L1.1 **First round:** By assumption A1, each node knows the identity and IP address of all its peers in its super-leaf and by assumption A2 these nodes are all reachable during the entire round (unless the node has failed). Recall that nodes use reliable broadcast to disseminate their state to all other nodes within the super-leaf. By assumption A4, messages sent by each node in this round are received without corruption by all other correct nodes in the super-leaf by the end of the round (if some nodes fail during the first round, they are excluded from contributing to the state of the super-leaf). Hence, all correct nodes receive the same set of messages by the the end of the round 1. Thus, based on the union property of state update:

$$\underset{l_{i,j}, l_{i,j'}}{\forall} \in s_i, \;\; \Pi(l_{i,j}, s_i, 1, 2) = \Pi(l_{i,j'}, s_i, 1, 2) \tag{14}$$

Note that by assumption A4 each correct node knows which other nodes have failed during the round. Once a node fails, it will not re-join other than through an explicit join message, based on assumption A2.

L1.2 **Second round:** In the second round, each node $s_{i,j}$ calculates the state of its height-2 ancestor $w = A(s_{i,j}, 2)$ by combining the states of the children of w, $C(w) = \{u_i\}$. Note that each node $s_{i,j} \in s_i$ is missing the state of the other vnodes $u_{i'} \in C(w)$ except u_i.

To collect the missing state, the super-leaf first elects k representatives from its set of correct nodes. We do not discuss this election process, other than noting that it is trivially accomplished using reliable broadcast.

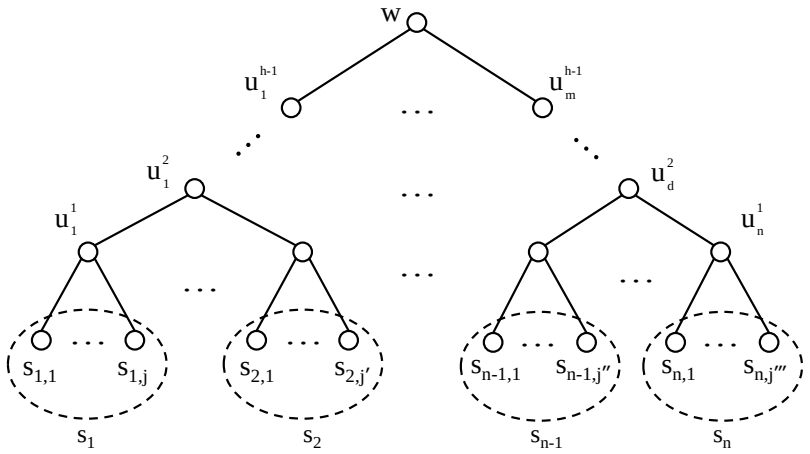


Figure 9: LOT of height $k+1$ with n super-leaves

Subsequently, each representative from each super-leaf s_i fetches state $\Pi(s_{i',j'}, u_{i'}, 1, 2)$, from node $s_{i',j'}$ where $i' \neq i$ and $s_{i',j'}$ is one of the emulators of $u_{i'}$ randomly selected from $T(s_{i,j}, u_{i'}, 1)$.

Based on assumption A1, all the chosen representatives have a valid emulation table and the same membership view at the start of the first consensus cycle. Therefore, the representatives already know the emulators of each vnode $u_{i'}$. However, the selected emulator may have failed since the last update of the emulation table. Hence, if the chosen emulator does not respond before a timeout, based on assumption A2 the representative assumes that the emulator has crashed, marks it as such, and picks another live emulator from the table. If no such emulator exists, the representative stalls.

In the best case, neither the representative nor any of the emulators fail during the first consensus cycle. However, it is possible for failures to occur, which complicates the process.

- We first consider the case when neither representatives nor emulators fail. By design, the representatives in super-leaf s_i receive $M(s_{i',j'}, \Pi(s_{i',j'}, u_{i'}, 1, 2), 1, 2)$ from each emulator $s_{i',j'}$ and use reliable broadcast to transmit it to every other node $s_{i,j} \in s_i$. Then, each such node uses Equation 5 to update its temporary state for w. In the absence of failures, all nodes in all super-leaves receive identical values for states of the vnodes u_i. Hence, all nodes in all super-leaves compute the same state for the w at the end of the second round, as desired.

- Now, suppose that one of the representatives for one of the super-leaves, say s_i fails to retrieve the state of some vnode $u_{i'}$. This could be due to a failure of the representative or the failure of the emulator, or message loss. Nevertheless, if at least one of the representatives assigned to fetch the state of each vnode $u_{i'}$ eventually succeeds, all the correct nodes in s_i will eventually receive $M(s_{i',j'}, \Pi(s_{i',j'}, u_{i'}, 1, 2), 1, 2)$ and complete the second round of the consensus cycle, as before.

- Now consider the situation when *all* representatives in super-leaf s_i fail in fetching the state of at least one vnode $u_{i'}$. The representatives may fail because either (a) all the emulators of $u_{i'}$ have crashed or are in a suspended state and no more emulators are available to respond to proposal requests for the state of $u_{i'}$ or (b) all the representatives in s_i have crashed or are in a suspended state, and this super-leaf does not have a sufficient number of live nodes to elect a new representative. In either case, the nodes in s_i will be unable to complete the second round of the consensus cycle and will stall and go into suspended state.

In all three cases, it is clear that nodes fall into three categories: (a) correct nodes that complete the second round and have the same state or (b) nodes that are stalled or (c) nodes that have failed. Nevertheless, all non-stalled live nodes have the same state, which proves Lemma 1. □

A.4.1 First cycle for a tree of height k+1

Consider a LOT of height $k+1$ consisting of n super-leaves, shown in Figure 9. The root-vnode w has m children of height k, i.e., $C(w) = \{u_1, u_2, \ldots, u_m\}$.

LEMMA 2 If, in a tree of height k, at the end of the k^{th} round in the first consensus cycle either(a) all correct nodes that complete the $(k)^{th}$ round compute the same state for the root node:

$$\underset{l_{i,j}, l_{i',j'} \in D(C(w))}{\forall} \Pi(l_{i,j}, w, k), 1, k+1) = \Pi(l_{i,j}, w, k), 1, k+1) \tag{15}$$

or (b) the consensus process is stalled for live nodes that do not complete the k^{th} round then in a tree of height $k+1$ at the end of the last round of a consensus cycle c, either (a) all correct nodes that complete the $(k+1)^{th}$ round compute the same state for the root node:

$$\underset{l_{i,j}, l_{i',j'} \in D(w)}{\forall} \Pi(l_{i,j}, w, c, k+2) = \Pi(l_{i',j'}, w, c, k+2) \tag{16}$$

or (b) the consensus process is stalled for live nodes that do not complete the $(k+1)^{th}$ round.

PROOF:

From the induction hypothesis, for the height-$(k+1)$ tree, correct nodes in super-leaf s_i that have completed round-k already have computed the state of their height-k ancestor $A(s_{i,j}, k)$. These nodes require the state of other children of the root vnode w, i.e., $u_{i'} \in C(w)$ where $u_{i'} \neq A(s_{i,j}, k)$, to complete the last round $h = k+1$. In round $k+1$, super-leaf representatives from all the n super-leaves fetch the state of node $u_{i'} \in C(w)$ from one of its emulators and use the reliable broadcast (Assumption A4) functionality to disseminate the state to live nodes in their own super-leaf. The representatives either succeed or fail to fetch the required state. We consider these two cases below.

L2.1 Representatives of super-leaf s_i succeed in fetching the state of each vnode $u_{i'} \in C(A(s_{i,j}, k+1))$, $u_{i'} \neq A(s_{i,j}, k)$.

The representatives of all the super-leaves fetch the same state of vnode u_i from its live emulators that completed the round k, based on the induction hypotheses. So all correct nodes $s_{i,j}$ receive the same of state of u_i, either fetched by themselves or received from the representatives of s_i. As a result, all the nodes that complete the last round $k+1$ compute the same state of the height $k+1$ root vnode w:

$$\underset{l_{i,j}, l_{i',j'} \in D(w)}{\forall} \Pi(l_{i,j}, w, c, k+2) = \Pi(l_{i',j'}, w, c, k+1) \tag{17}$$

which proves the desired property.

L2.2 None of the representatives of a super-leaf s_i succeed in fetching the state of at least one vnode $u_{i'} \in C(w)$.

The representatives of a super-leaf s_i can fail to fetch state of $u_{i'}$ because of one of the following two reasons:

1. All the representatives of super-leaf s_i have crashed and s_i does not have sufficient number of live nodes to elect a new representative. In this situation, the consensus process stalls for s_i in the current cycle c.

   Nodes in other super-leaves may succeed in fetching the state of $A(s_{i,j}, k)$ from the live descendants D($A(s_{i,j}, k)$) of $A(s_{i,j}, k)$ and complete the last round $k+1$. Following the same reasoning as given in the previous case in L2.1, these nodes compute the same state of the root vnode w as desired.

2. The representatives of super-leaf s_i are alive but they fail in fetching the state of $u_{i'}$. This can happen only if all the nodes descending from $u_{i'}$ have not completed the round k either because (a) all the descendants of $u_{i'}$ are stalled before completing the round k or (b) because all the descendants of $u_{i'}$ have failed. In either case, the consensus process stalls for the nodes in s_i in the current cycle c. Other nodes that succeed in fetching the state of $u_{i'}$ may complete the last round $k+1$ and compute the same state of the root vnode w, which follows from the same reasoning as given in L2.1.

L2.3 As the two cases in L2.1 and L2.2 are exhaustive, this proves Lemma 2. □

A.4.2 Subsequent cycles

LEMMA 3: For a tree of height k, if at the end of consensus cycle $c-1$, either (a) all correct leaf nodes that complete the cycle have the same state for the root node:

$$\underset{l_{i,j},l_{i',j'} \in L}{\forall} \Pi(l_{i,j}, A(l_{i,j}, 2), c-1, 3) = \Pi(l_{i',j'}, A(l_{i',j'}, 2), c-1, 3) \tag{18}$$

or (b) the consensus process is stalled for some live nodes, then at the end of consensus cycle c, either (a) all correct leaf nodes that complete the cycle have the same state for the root node:

$$\underset{l_{i,j},l_{i',j'} \in L}{\forall} \Pi(l_{i,j}, A(l_{i,j}, 2), c, 3) = \Pi(l_{i',j'}, A(l_{i',j'}, 2), c, 3) \tag{19}$$

or (b) the consensus process is stalled for some live nodes.

Proof: If all live nodes are stalled in cycle $c-1$, then cycle c cannot be initiated, and in this case the lemma is trivially true. Hence, we only consider the case when consensus cycle $c-1$ terminates with only some stalled nodes, with all live nodes having the same state for the root node.

If some live nodes were stalled at the end of cycle $c-1$, these nodes are unable to participate in the first round of cycle c, and this prevents the computation of the state of their super-leaf vnode parent. Thus, all other live nodes in all other super-leaves will also be stalled in the second round of cycle c, which completes the proof of the lemma.

Note that the only other difference between the first consensus cycle and cycle c is that at the start of the first consensus cycle, the emulation table at all live nodes is known to be identical and accurate. This emulation table may be updated in subsequent cycles, since some nodes may have failed (and recovered) during the first $c-1$ cycles, thus introducing potential inaccuracies. Hence, to prove this lemma, we only need to show that these inaccuracies do not affect the correctness of the consensus process.

Recall that the structure of the emulation table is that it maps from a vnode n to its set of potential emulators $\{s_{i,j}\}$. If the emulation table is inaccurate, one of three cases must be true:

1. $s_{i,j}$ is in fact not a descendant of n
2. $s_{i,j}$ is a descendant of n but is incorrect (either crashed or stopped)
3. $s_{i,j}$ is a live descendant of n but is not marked as a potential emulator in the emulation table

Note that the first and third cases are impossible due to the design of the protocol (see Definition 15). The second case will lead to a representative who uses this emulator to fail to get a proposal-response. However, this is identical to the failure of the emulator or representative, as discussed in Lemma 1, and hence does not affect the correctness of the protocol.

Since these cases are exhaustive, this proves Lemma 3. □

A.4.3 Proof of Theorem 1

Proof: The proof follows by induction from Lemmas 1, 2, and 3. □