IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 11 June 2024, accepted 24 June 2024, date of publication 1 July 2024, date of current version 9 July 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3421582

RESEARCH ARTICLE

LowPaxos: State Machine Replication for Low Resource Settings

ALEX MWOTIL[1], THOMAS ANDERSON[2], BENJAMIN KANAGWA[1], THEANO STAVRINOS[2], AND ENGINEER BAINOMUGISHA[1]

[1]College of Computing & Information Sciences, Makerere University, Kampala, Uganda
[2]Paul G. Allen School of Computer Science, University of Washington, Seattle, WA 98195, USA

Corresponding author: Alex Mwotil (alex.mwotil@mak.ac.ug)

ABSTRACT State Machine Replication (SMR) is a popular framework for building highly available fault-tolerant systems, and widely uses distributed consensus as an implementation approach. Consensus protocols usually require a leader replica to coordinate the actions of other members in ensuring that the sequential command log to retrieve and update the state of the system is consistent despite failures. These protocols often assume homogeneity of the deployment environment and resource capabilities of all replicas each with an equal chance of leadership. However, challenged environments are mostly heterogeneous and the choice of a good leader can offer better performance. This paper introduces LowPaxos as a distributed consensus protocol for challenged environments. LowPaxos uses the computing and network capabilities of the replicas, and operational environment to designate the best leader, and adapt accordingly to changes characteristic of these settings. LowPaxos is evaluated against both leader-based (MultiPaxos) and leaderless (EPaxos) protocols and demonstrates performance gains of upto 5X and 2X respectively in a heterogeneous challenged setting.

INDEX TERMS State, replication, resource-constrained, challenged, low resource, consensus, distributed.

I. INTRODUCTION

State Machine Replication (SMR) is a popular framework for constructing fault-tolerant and highly available distributed systems. Given that the system components (clients and replicas) often reside in disparate domains, they are susceptible to various failures, including network communication breakdowns, as well as hardware and software outages [6]. Distributed consensus has emerged as one of the main approaches for implementation of SMR and has attracted vast research interest for nearly four decades [1]. In the context of SMR, replicas interact to maintain a synchronized command log of client requests that modify or retrieve the system's state. These commands are executed in a deterministic and sequential order on every replica, ensuring a coherent system state despite failure. Typically, the ordering of the log is coordinated by one of the replicas designated as a command leader, that contacts a quorum of replicas as per the semantics of the protocol. These leader-based protocols are prevalent in most production systems due to their simplified design and ease for recovery under failure [14]. A number of services adopt the the SMR model including Apache Zookeeper [4], Chubby [3], Redis [5] and etcd [2] to attain consensus and offer a coordination function for a diverse range of applications.

Leader-based protocols often assume homogeneity of the deployment environment and resource capabilities of replicas assigned to the leader role. Replicas have an equal chance of assuming leadership, either through an election process or by being the first point of contact with clients. This is generally acceptable if each replica is able to perform and sustain the leader function while meeting an application's Service Level Objectives (SLOs). However, heterogeneous environments such as those in the developing world usually have non-uniformity in processing and network capabilities. Leaders have to communicate with a majority of replicas for consensus, and hence their position and distance to other replicas in the distributed system can significantly impact performance,

The associate editor coordinating the review of this manuscript and approving it for publication was Rahim Rahmani.

particularly for latency-sensitive applications. Additionally, the leader serves as a traffic concentration and reference point, and factors like its availability and processing power are important considerations. Ideally, leaders should be at the apex of the performance pyramid, more reliable/available and have the least latency communicating with a quorum of replicas required for consensus. The leader-based protocols with homogeneous assumptions will perform at the speed of the leader replica in a heterogeneous setting, which may be the slowest in the system.

Leaderless or multi-leader protocols, such as Egalitarian Paxos (EPaxos) [7], are primarily designed for heterogeneous Wide Area Networks (WANs) to enhance throughput by reducing commit latencies for non-interfering requests or commands. This is achieved because the command leader advancing a client request needs to contact only a minor quorum of replicas for commits using a fast-path mechanism. However, the performance of a leaderless protocol degrades to that of a leader-based protocol when conflicting commands predominate. In such cases, the command leader must contact a majority of other replicas in a typical two-phase (slow-path) cycle to reach a decision. In addition, each leader must maintain a dependency tree of commands, which can grow exponentially in high-load environments, thus requiring more computing resources on each replica. It is assumed that each replica can lead the state machine's operations, but in some deployments, certain replicas may be too slow to effectively fulfill any command leadership role.

Challenged environments or low resource settings make this problem worse. These environments often have resource limitations in terms of processing and network capabilities. Network failures are more common, and network performance can be erratic. Replicas may exhibit varied and degraded periodic performance. A leader may be a good choice at one point in time, and not some time later. To better understand this environment, a 5-node distributed cluster is set up in various cities in Africa. The nodes run Paxos, a leader-centric distributed consensus algorithm, for approximately 24 hours and a client configured to issue requests over this duration. Figure 1 shows the 99*th* percentile latency plots for Paxos. While the optimal (minimum) latency is typically 72*ms* over this duration, there are periods of significantly higher latency. The nature of challenged environments dictates that conditions can shift unexpectedly. These conditions usually necessitate leader role changes as long-term replica leadership can affect performance. The inability of Paxos to detect and adapt to these changes can further drain application performance. It is important to consider and adapt to the dynamic properties of a challenged environment in the design of SMR protocols for these settings.

This research introduces LowPaxos, a *'strong leader'* adaptation of the Paxos consensus protocol designed for challenged environments. The protocol uses the performance attributes of the replicas to determine the best leader and continuously evaluate and adapt to the dynamic conditions

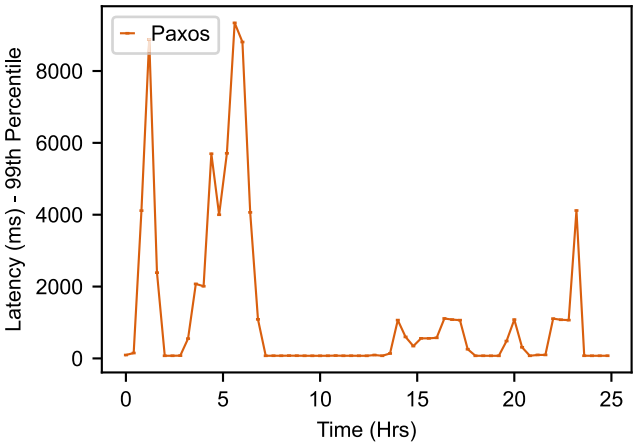

FIGURE 1. The 99*th* percentile latency plots of Paxos for a approximately a 24-hour duration. The optimal (minimum) latency is typically 72*ms* over this duration but there are periods where this is significantly higher. Challenged environments exhibit dynamic behavior especially in the network properties and the inability of Paxos to detect and factor these in leadership can degrade application performance with some results indicating higher latencies of upto 8500*ms* over some request-response cycles.

of these environments. LowPaxos represents the relative strength of a replica using a profile - a value computed based on its performance attributes such as link capacity, network latency and packet loss against other replicas and their weights as assigned by a system operator. Other attributes include the replica's probability of availability and speed (processing power) based on response time and available resources. The election of a strong leader closely follows the quorum consensus logic. A candidate replica contacts a majority of other replicas with profile information asserting its suitability for the leadership role. Other replicas will vote if the profile of the candidate is superior to the local replica profile. If successful, the replica leadership lease is maintained within the system consensus operations and additional election logic. For example, a group of *rebel* replicas are primed for leadership and will monitor the performance of the leader. Other replicas designated as witnesses also receive heartbeats from the leader. If the performance drops below a defined threshold, a new election round is initiated. In the best case, no election is required if the current leader is still strong. In summary, LowPaxos aims to:

- Establish a strong leader for the replica system based on a profile-driven election.
- Adapt the operation of the replica system to dynamic changes of challenged environments.
- Improve the overall performance of the system by increasing the throughput and minimizing latencies for operations in an heterogeneous setup.

LowPaxos strives to attain performance gains, including throughput and latency, compared to both leader-based (up to 5X improvement for MultiPaxos) and leaderless (up to 2X improvement for EPaxos) protocols in a heterogeneous challenged setting. The remaining sections of this paper are

organized as follows: Section II provides a background for the study, Section III contains related work, Section IV describes the LowPaxos protocol, Section V presents the evaluation of LowPaxos in comparison to other protocols and Section VI concludes the paper. The terms 'replica', 'node', 'server' and 'component' shall be used interchangeably.

II. BACKGROUND TO THE STUDY

Fault-tolerance is an important design characteristic of production systems today. This is because large scale systems are inherently vulnerable to faults, that range from software bugs to communication and hardware malfunctions. The components of such systems are often distributed across various geographical locations, thereby expanding the fault domain. Even within the same locale, the components are not immune to failure. Consensus protocols play a key role in ensuring both high availability and correct operation of distributed systems especially in the face of failures. This allows for development of complex applications built on top of consensus, using the Replicated State Machine (RSM) abstraction. At the application level, RSM operates without concern about failures of supporting components. The following subsections provide an overview of consensus algorithms and highlight adaptation issues for challenged environments.

A. CONSENSUS ALGORITHMS

Consensus algorithms involve a group of nodes or processes coordinating to agree on a sequence of actions and the final state of a system, even in the face of potential failures such as delayed, reordered, or lost network messages, as well as node failures. Given an ordered sequence of operations or commands, the nodes communicate to propose, agree, and commit to a specific value or decision, while upholding the safety and liveness properties of the protocol. One or more of the nodes in the system will act as the leader and is tasked with advancing commands towards agreement and subsequent execution. As these protocols have evolved, there has been a shift in design focus towards optimizing performance, aiming for low latency and high throughput, and adapting to the context, whether it's a data center, a local area or wide area network. Consensus protocols find applications in various domains, including distributed databases and file systems. RSM builds on consensus protocols to ensure that distributed processes, or replicas, maintain an identical state. This synchronization enables client operations to read from and update a consistent state. Leslie Lamport's Paxos is one of the most renowned consensus protocols owing to his seminal work in this field [36].

Paxos designates *proposer*, *acceptor*, and *learner* roles to replicas to service client requests. It also defines two phases: Phase 1 (Prepare) and phase 2 (Accept). The client sends a request (command to retrieve or update the system state) to one of the replicas. The recipient replica will act as proposer (leader) for this request and will send proposal messages for each request (or request batch) to the rest of the replica members (acceptors) as shown in Figure 2. Unique identifiers are assigned to messages for liveness and conflict resolution. The success of the leader request will depend on the number of acceptors that concur with the proposal. The leader requires responses from a majority of the acceptors (including itself) before it can propose a value or request a commit of an operation. This is the prepare phase of the protocol. As with the proposal request, the leader needs a majority of responses from the acceptors for the proposed value (commit message) in order to provide a response to the client. This is the accept phase of Paxos. Other replicas that may not have been part of the decision will eventually learn in subsequent operations. Paxos guarantees of liveness and safety of a RSM are formally verified.

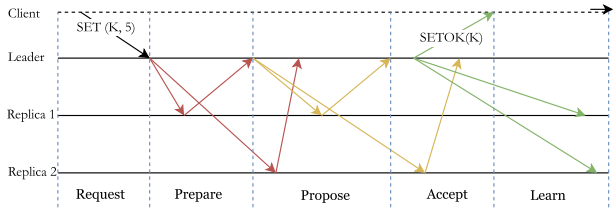

FIGURE 2. Normal operation of the Paxos protocol on a three-node system. The leader replica is in charge of moving the request through the different stages until a response is provided to the client. For each message sent to the replica members and corresponding responses, the leader requires a majority in order to maintain both correctness liveness and safety.

For each client request, Paxos requires completion of the two phases (and hence 2 Round Trip Times (RTTs)). The prepare phase of Paxos is typically being used for election of a leader for a given request. Optimizations of the Paxos protocol such as MultiPaxos introduce a stable or distinguished leader that can propose multiple values without the need for a rerun of the prepare phase. The leader can be elected in advance or on receipt of an initial client request. As with other replicas, the leader may fail, and if so, a new one should take over the role while ensuring that consistency is not compromised. When the old leader rejoins the system, the state of the system could have advanced and will need to learn other previous operations through state transfer from updated replicas. If stable storage is used, only a portion of the state is requested by the replica. In an heterogeneous setting, the choice of the leader is important as its speed and latency to reach other replicas can affect the performance of the system. In Figure 2, the leader and Replica 1 are more strongly connected compared to the leader and Replica 2. The network latency between the leader and Replica 1 is lower than that between the leader and Replica 2, since Replica 1 receives and responds to the message first. It is also possible that the performance of the system (throughput) is higher with Replica 1 as the leader. Hence, the decision on which replica should be the leader is important.

For most consensus protocols, resource considerations are often only partially addressed or, in some instances, entirely overlooked. This might be acceptable in environments where

operators exert more control over infrastructure and nodes exhibit a high degree of homogeneity. However, the growing trend towards edge-driven and microservice applications introduces some form of heterogeneity to distributed system setups. In these settings, determining which replica should assume the role of leader is a critical decision. Are there specific resource-related factors that need to be considered when choosing a leader? Moreover, how should the system react when resource parameters experience variability? Additional insights are provided in Section II-B with a reference environment of a low resource setting.

B. LOW RESOURCE SETTING

Low resource computing environments (challenged or resource-constrained) are faced with technological and resource challenges [18] in data and compute (resource quality and quantity) and network (network partitions, capacity and cost limitations) that complicate the design of high performance and available distributed applications. Public cloud computing providers such as Amazon, Google and Microsoft have no infrastructure in most resource constrained environments [19], [20] due to setup costs, demand, market dynamics and other contextual challenges. Consequently, there is high reliance on low resource data centers that are unreliable due to power and network outages, run old hardware and have limited processing power. Even with better resources, network communication between locations is inconsistent. In other cases, performance is sacrificed for availability by entirely relying on distant dense resource pools of the public cloud. More specifically, these properties (R) include available compute resources (CPU (C), Memory (M) and Disk (D)), network (throughput (T), packet loss (J) and latency (L)) and the network reachability (A) of a node. Collectively, these properties can affect the design and operation of distributed systems that serve clients or users in these settings.

In addition, the operational environment of a low resource setting is dynamic. The network properties are variable, and in the extreme case the network is unavailable and hence the reliant nodes. A profile is the resource strength of a node to lead a group of other replicas, with a higher value over others denoting its superiority. It is a sum of the weighted properties of a node's resources and network communication metrics. A node performs a series of network measurements against other node members and records the throughput, packet loss and latency values. The nodes additionally probe each other to determine the availability/reachability status. Each of these attributes is weighted as per its importance to the operation of the system. The profile of a node against another is computed as shown in Equation 1. For a 5-node cluster located in different cities in Africa, a series of network, computing resource and availability measurements are performed and profile computations recorded. Figure 3 shows the profiles of the other 4 locations relative to the Dar-es-Salaam node plotted over a 24-hour period. For example, the performance

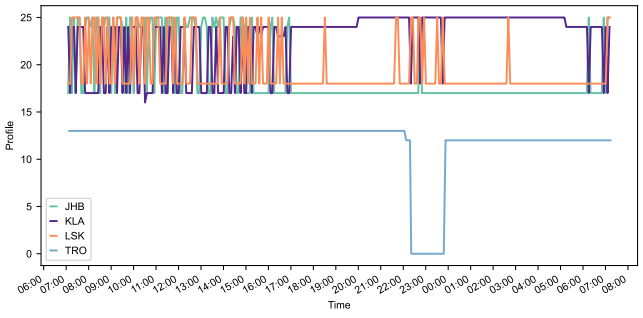

FIGURE 3. The profile evolution of locations JHB, KLA, LSK and TRO with respect to DAR over a 24-hour period. The profile is computed according to Equation 1 with a higher value indicating superiority. The performance of TRO is worse compared to the other three locations. For some locations, the profile is rather uniform but for most it is usually variable. The leader may vary over time in a leader-based SMR protocol for these environments.

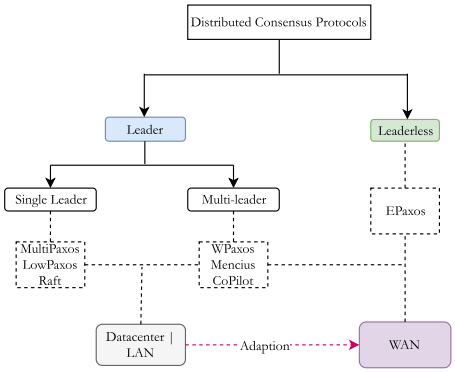

FIGURE 4. Related Work - Distributed consensus algorithms can broadly be categorized into leader or leaderless (multi-leader) variants and usually are designed for specific network environments.

of TRO is much worse in comparison to KLA, JHB and LSK. The heterogeneity of challenged environments due to hardware, data center and network attributes leads to the profile variations shown and validates the need for property-based election of a strong leader, and its subsequent configuration of its desirable replicas for quorum.

Weighted profiles have been used for leader election before under different contexts and for different use cases: ring networks [21], Software Defined Networks (SDNs) [22], Swarms [23], [24], Asynchronous communication networks [25], Wireless sensor networks [26], [27], Internet of Things (IoTs) [28], [29], Security [31] and personal distributed environments [30]. LowPaxos uses weighted profiles to determine the initial distinguished leader replica, and continually assess its performance and capability to sustain this function. The leader replica is changed when its performance degrades below a defined threshold.

III. RELATED WORK

Egalitarian Paxos (EPaxos) [7] is a leaderless protocol that uses command interference properties to determine the execution path of a command in the wide area replication systems. It defines two paths: slow and fast. The slow path

takes the typical Paxos *Propose-Prepare-Commit-Accept* phases for conflicting commands. The fast path improves performance by requiring only *(f + (f + 1)/2)* of the replicas to agree on the command slot. EPaxos delivers optimal commit latencies in a WAN in some cases, and can exhibit worse behavior than centralized approaches if: (i) there is a high percentage of command conflicts and (ii) the command requests are sent from diverse locations. MultiPaxos can outperform EPaxos in latency under high conflict rates [8]. In LowPaxos, a better centralized leader can provide even better performance especially for diverse and conflicting requests.

MultiPaxos (MPaxos) [9] and LowPaxos rely on an elected leader to coordinate actions. More specifically, Paxos and its derivatives such as Viewstamped Replication (VR) [37], [38] take the *blind* leader approach to progress the RSM. Clients send command requests to any of the replicas from which proposals for command slot leadership are advanced. The replica contacted could be at the tail of the latency chain amongst the participants. In addition, the progress of the RSM is dependent on 2 Round Trip Times (RTTs) between the command leader *proposer* and a majority of the members. This can potentially affect the performance of the system and in the worst case, the RSM operates at the speed of the slowest replica. In as much as both rely on the single leader, it is imperative to make strategic decisions on this position. In addition, LowPaxos considers the capabilities of the replicas and designates responsibilities accordingly. Raft [10] decomposes the consensus problem into leader election, log replication and safety for better understandability. It additionally provides for cluster reconfiguration and log compaction. LowPaxos is partly an optimization of Raft with capability assignment of roles to the replica members as dictated by the *strong* leader. LowPaxos introduces a a new form of leader election based on the conditions of the operating environment.

In Mencius [11], the WAN replicas are partitioned to lead specific instances of the replicated log for improved overall performance of the system. The assumption is that all the replicas have ability to lead and advance requests for their collocated clients. It is also expected that the local replicas continue operating normally for the most times. The local availability of the system is tagged to the operational status of the local replica. In a low resource setting, there is additional complexity in managing state and conflicts given the resource disparities of the replicas. The load-balancing features of Mencius are partly vested in the role designation function of LowPaxos. The assumption is that the *rebels* perform equally important consensus decisions of the RSM as potential leaders. To further provide for shared load, one approach is to shard the leader role. This is most appropriate for much wider area networks with better and more disparate resource footprints, and is planned for future work. The operations can be classified as per their requirements, for example introduce different leaders for CPU and network intensive operations.

In CoPilot [12], replicas exhibit heterogeneous behavior in which slowdowns are possible due to changes in the network and host-related issues. It is a multi-leader (pilot and copilot) variation of EPaxos [7] and MultiPaxos (MPaxos) [9]. The complexity of maintaining additional dependency state between the pilot and copilot could potentially affect the performance of the RSM. In addition, the evaluation of CoPilot does not consider a WAN environment where clients could be collocated with the slowest replica. LowPaxos builds on the CoPilot heterogenous behavior of replicas to determine the *faster* long-term leader and subsequent roles of the remainder replicas relative to the leader resource strength. In most cases, the slow replicas are at the tail end of the ordered configuration.

Other RSM protocols such as NoPaxos [16] and Spec-Paxos [15] require considerable control over the participants or the underlying communication channel, and provide significant performance improvement in the data center. Flexible and variable quorum-based protocols such as FPaxos [13], EdgePQR [17] and WPaxos [14] have also been advanced - these are classical Paxos algorithms with quorum relaxations at the different phases of Paxos. The implementations require multiple replicas at the WAN (zone) sites that form the major quorum part of consensus for client requests originating from the same location. LowPaxos considers environments with limited edge resources to run multiple replicas of the system. In addition, there is little control over the environment and communication is mostly over the asynchronous Internet.

IV. LOWPAXOS

LowPaxos is a derivative of Paxos that specially draws inspiration from the Viewstamped Replication (VR) [37], [38] variant of MultiPaxos. Monitoring, leader election, reconfiguration and recovery are integral elements of the protocol as shown in Figure 5. This section provides a description of LowPaxos and its consistency quarantees in challenged environments.

A. PROFILES

The profile is a cumulative property of the weighted attributes of a pair of nodes in the system and the network environment. For a set of nodes (n) in different locations of a low resource setting, the *profile* for a pair of nodes (i & j) is computed as shown in Equation 1.

$$P_{i,j} = \sum_{i,j=1}^{i,j=n} W_R.R_{i,j},\ P_{i,j} = 100\ \forall (i = j) \tag{1}$$

$$R \in (Compute, Network, Availability) \tag{2}$$

$$0 \leq W \leq 1, 0 \leq P_{i,j} \leq 100 \tag{3}$$

For Compute between a pair of nodes;

$$R_{i,j} = \sum \frac{R_i.W_R}{\sum (R_i, R_j)}, R \in (C, M, D) \tag{4}$$

For the Network, the ideal throughput should be maximal while latency and packet loss should be minimal. For all

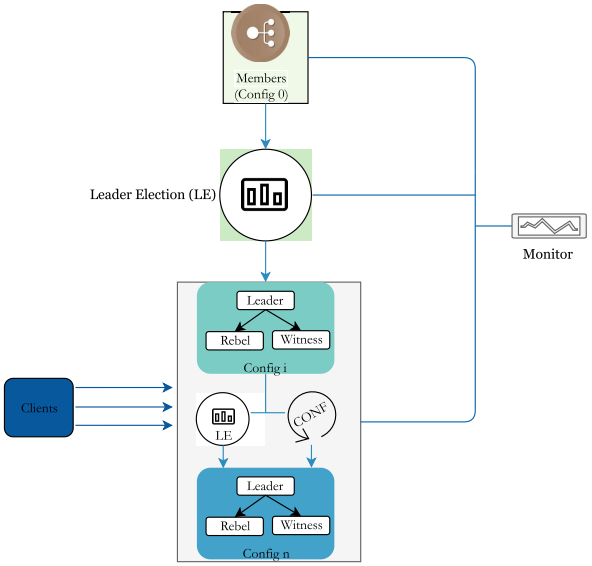

FIGURE 5. LowPaxos is composed of the monitoring, leader election, configurator and the distributed consensus algorithm. Monitoring provides the profile information required for elections and role distribution by the configurator.

network measurements recorded, the maximum throughput (T*max*, minimum latency (L*min*) and minimum packet loss percentage (J*min*) are recorded. The network profile between a pair of nodes i and j is then computed as follows (Equation 5):

$$N_{i,j} = \sum W_R.(\frac{R_{i,j}}{R_{max}} \vee \frac{R_{min}}{R_{i,j}}), R \in (T, L, J) \quad (5)$$

On availability, each node keeps track of uptime values for each member. For a number of values k, the mean availability value is computed as shown in Equation 6. The availability, A, of a node indicates its uptime probability with respect to another member.

$$A_{i,j} = \frac{(\sum_{n=1}^{n=k} A_n)}{k}, 0 \leq A_n \leq 100 \quad (6)$$

Equations 4, 5 and 6 provide computing resource, network resource and availability profiles of a given node against another in the distributed system. These profiles may be aggregated by percentiles.

Table 1 outlines the symbols used and their short descriptions.

The compute resource consumption of a node and network measurements to other nodes is pushed to a *Monitor* replica. The assumption is that this replica is highly available to all nodes. The profile computations are managed by the monitor and retrieved by a node when required. For improved performance, profiles may be cached for a defined period of time before being marked as stale. In this case, a node will request for new profile information from the monitor.

TABLE 1. Description of symbols used.

No	Symbol	Description
1	C	Available CPU resources for a node based on mean utilization
2	M	Available Memory for a node based on mean utilization
3	D	Available disk space for node based on current utilization
4	$T_{i,j}$	Throughput between node i and j
5	$L_{i,j}$	Mean latency between node i and j
6	$J_{i,j}$	Mean packet loss percentage between node i and j
7	$A_{i,j}$	Availability of cluster j from i
8	W_r	Weight of a resource r
9	k	Total number of values of a metric
10	R_{min}	Overall minimum (target) resource value in the system
11	R_{max}	Overall maximum (target) resource value in the system
12	$R_{i,j}$	Compute resource profile of a node i wrt j
13	$N_{i,j}$	Network resource profile of a node i wrt j
14	$P_{i,j}$	Normalized overall profile of a node i wrt j

B. DESIGN AND IMPLEMENTATION

A LowPaxos replica has a unique identifier in the system and maintains internal state for both leader election and consensus operations of the state machine. Each replica has a *ballot(x, y)*, a tuple that contains the leader term *(x)* and the next operation slot *(y)* numbers. The leader term is predominantly used for election-related functions. The slot number is used to propose the ordering of a client request. A new term signifies a successful election cycle or leader assertion. In leader assertion, a replica affirms its strong case to maintain its leadership role in case it has been incorrectly marked as offline or with a degraded profile. The ballot is incremented to indicate a new leader term or client request that needs to ordered in the log. A ballot is fresh if a replica can confirm that the term is known and the slot number is the next expected. The ballot ensures that replicas are acting on fresh information and hence guarantee progress of the state machine. Replicas maintain additional state including leader and vote information of the last election, local replica status and the role played at a given point in time.

The replica is initialized as a member and must transition to one of the operational roles: *leader*, *rebel*, or *witness* in order to process requests. After a successful election, the new leader sorts the remaining replicas based on their profiles with respect to it and assigns roles. These roles are part of a configuration sent to the members. The leader role involves overseeing all consensus operations and providing responses to clients. A rebel replica is poised to assume leadership if the leader's profile degrades or if the leader becomes unreachable. Rebels generally maintain synchronization with the leader regarding state information, and are usually more advanced in commit and execution actions. Witnesses actively participate in elections and log operation ordering but do not execute operations on the state machine. LowPaxos defines five distinct modes for a replica's operation: *INITIALIZATION*, *ELECTION*,

CONFIGURATION, *RECOVERY*, and *NORMAL*. The default mode for a replica is *INITIALIZATION*. Replicas engaged in leader voting processes switch to the *ELECTION* mode. When a replica detects that its leader information is outdated, as indicated by the ballot in one of the received requests, it transitions to *CONFIGURATION* and requests updated information from the new leader that contacted it. The new information typically necessitates the replica to additionally request for new state from the leader, and hence will first transition to *RECOVERY*. The ultimate and fully operational mode for a replica is *NORMAL*, where it can process requests.

A configuration (*Config(ballot, n, f, Reps)*) in LowPaxos contains the ballot, the number of replicas *(n)*, allowable faults *(f)* and replicas information (*Reps*). The ballot is used to guarantee the freshness of the configuration message. The Reps (*id, role*) is a vector of replicas with their identities and the roles as ordered by the leader. A replica utilizes this role information to discern its capabilities in the context of the consensus operations within the system. The replica-specific information is also used to calculate the required quorum size for consensus. At the inception of the protocol, each replica is provided with an initial configuration denoted as *Config 0*. When a new leader is elected, it generates a configuration *Config n* that it shares with the rest of the replicas. Furthermore, a leader can create a new configuration if it is erroneously marked as offline or incorrectly deemed as having a lower profile compared to the rebels. A replica may also explicitly request a new configuration if there are higher terms than known in new messages received. The configuration may introduce role transitions, for example a witness replica promoted to a rebel. In this case, the witness replica will have to execute all previous operations while optionally requesting for an updated state from the current leader.

Consensus protocols leverage timeouts for various purposes, including elections, request processing, and failure detection. In the context of elections, LowPaxos employs timeouts to initiate the default or initial election process within the replica system. Additional timeouts are utilized to ensure that an election eventually results in a new leader. The leader, once elected, operates with a term lease timer to periodically provide liveness updates to the other replicas. Conversely, a rebel or witness employs a heartbeat timeout, within which it anticipates communication from the leader. A rebel maintains an additional poll timer to proactively verify the availability status of the leader replica. If no response is received within the heartbeat or poll timeouts, a replica will flag the leader offline. As part of its liveness and assertive assurance to the remaining replicas, the leader includes profile information. The profile information is used to periodically assess the performance of the leader and its continued leadership role. The timeouts associated with the leader status are reset upon receiving any message from the leader. The remaining timeouts are primarily used for processing of client requests.

Each client request has a unique request identifier assigned by the client. Upon receiving a request, the leader assigns an operation number. Each replica maintains a log vector indexed by the operation number, which can begin at any arbitrary value. This log comprises entries corresponding to requests received by the replica, either directly from clients or as proposed by the leader. Each entry is uniquely identified by the slot number and includes the request, along with an optional response. The status of an entry can be one of the following: *Request* (the initial status upon client request receipt), *Propose* (indicating the request has been logged, and the leader is proposing it to other replicas), *Proposed* (signifying that one or more replicas have acknowledged the leader's proposal), *Committed* (a quorum of promises has been received at the leader and the replica has received the commit message from the leader), or *Executed* (the request has been executed on the state machine). When the request is executed, the result is logged, and a response provided to the client.

The replicas in LowPaxos communicate via message passing over User Datagram Protocol (UDP). A message includes metadata (wrapper) that specifies its type, for example, whether it is a commit or a vote request message. Depending on the type, additional information is included as per the semantics of the message. This message is then serialized and sent over the transport channel. The transport channel provides for point-to-point and broadcast messages. For internal replica communication (such as timeout triggers), Multiple Producer Single Consumer (mpsc) is exclusively used to transmit messages. For message processing, LowPaxos defines two message channel types: Network (UDP) and TX (mpsc). When a network message is received on the local buffer, the replica mpsc transmitter sends the message to the mpsc receiver. For network messages, the receiver retrieves the wrapper and message contents and forwards to the handler for further processing. Messages invoke handler operations at the receiver and are of election, consensus or liveness types.

Algorithm 1 describes how LowPaxos works. LowPaxos is implemented in Rust with ≈7K Lines of Code (LoC) with ≈1.8K for leader election and ≈5.2K for consensus operations.

1) NORMAL OPERATION OF LOWPAXOS

The normal operation of LowPaxos proceeds as shown in Figure 6. LowPaxos is a leader-based protocol and will require that a strong leader has been elected and is available before requests can be processed. The replicas use their ballots, roles and peer profile information from the *Monitor* to request for votes. A candidate changes its status to *ELECTION* and sends a *RequestVote(ballot, profile, role, type)* message to each of the members. The role is used to validate the type of election that a replica has initiated. LowPaxos defines five types of elections:

- **Default**: When replicas start, a random timer is initiated. At the expiry of this timer, a replica issues the vote

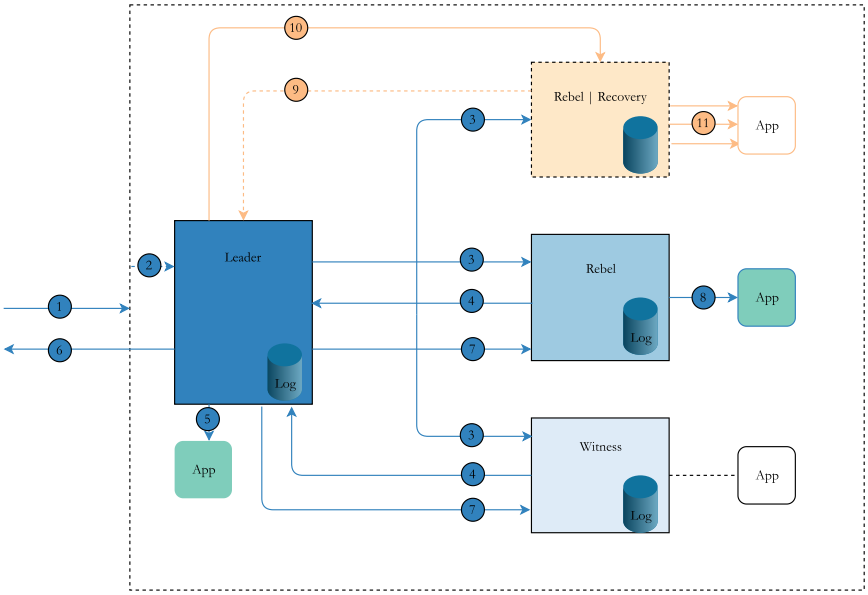

FIGURE 6. The normal operation of LowPaxos: (1) A client sends a request to the replica system which eventually ends at the leader (2). The leader increments the slot number in the ballot, logs the request and sends a proposal to the replica members (3). The members validate the proposal and provides a response to the leader (4). The leader needs a quorum of responses to commit the operation and subsequently execute it on the application (5). The leader will then provide the result of the execution to the client (6) and sends the commit message to the members (7). The members will update the log accordingly and the operation is executed (8) by only the rebel replica members. Using the ballots, the replicas my get outdated and will need to request for new state from the leader (9). The leader will provide a section of the state as per the request to the most recent commit (10) that will be executed by only the rebel replica (11).

request. On receipt of this request, a replica confirms the freshness. A replica will provide a vote response if the profile of the source replica matches the local value or is better. Otherwise, the replica will temporarily change its role to a *rebel* and start a timer within which it will either expect to receive leader information, or start a new *timeout* election type. Replicas that have voted will also expect leader information within a timeout period before ultimately starting the *degraded* election type. The timeout election takes precedence and hence the timer is shorter. The randomness of this election type reduces possibilities of conflicts at the start of the protocol and is similar to Raft [10] in operation. The main role of the default election type is to trial the possibility of electing a leader within the first round as a *fast path*. While this may not be achieved, other election types provide more opportunities for electing a leader.

- **Timeout**: Only a *rebel* can start this election type after an unsuccessful *default* election type. The chances of electing a leader at this stage increase as there is at least a replica that will provide a vote response. A similar or newer ballot term is allowed for vote processing. The recipient replicas check this ballot property and subsequently the profile information. As before, new rebel replicas may be created that could potentially restart this election type or a vote response is provided to the source replica.
- **Profile**: As replicas communicate in the lifetime of an application, profile variations are a norm in challenged environments. The current leader performance may degrade and is not the best choice going forward. A threshold is defined on what is an unacceptable degradation in the performance of the leader. Profile information is shared in two ways: the leader includes it in *heartbeat* messages sent to all replicas and in *responses* to status checks (polls) by the rebels. The recipient replica compares the profiles and if outside the threshold will initiate the profile election type. If a leader receives this election type request and confirms that its profile is indeed degraded, it will change its mode to *ELECTION* and provide a vote response. The leader also sets a timer within which the rebel is expected to have attained leadership. If the leader doesn't receive new leader information or still has a better profile, it will *assert* its leadership role by increasing the term in the ballot and sending the new *configuration* to the rest of the replicas.

- **Offline**: As with the profile election, the leader maintains its liveness position by providing regular heartbeats and poll responses to the replicas. The leader defines a lease timer after which it broadcasts the heartbeat message. The replicas also define the heartbeat timer within which they will expect to hear from the leader. The rebels take a step further: poll the leader at much shorter intervals and expect a response. General heartbeat messages from the leader reset the poll timers. The leader will assert its leadership position if it receives this election type.
- **Degraded**: The degraded election mode is provided to ensure progress in the election of the leader. This is similar to other leader-based consensus protocols as the profile information is not used in an election. Any replica in the system has an equal opportunity for leadership given it has the most current state. LowPaxos will use the profile election type at some point of the transaction to ensure that a better leader is eventually elected.

A LowPaxos client has a unique identifier and keeps track of its request and leader information. The client is started with the default configuration *Config 0* and will send the first request to a random replica. The leader replica only provides responses to client requests. For each response received, the leader information may be updated at the client. The request information is a vector of requests each with a status flag and the instants at which it was sent and response received. The client sets a timeout for each request after which it will be resent to the leader or another random replica. The client assigns a monotonically increasing identifier, generates the payload as *REQUEST (id, operation)* and sends it to the replica system. The rest of the request processing proceeds as follows:

1) The leader receives the request and proposes it to the members. If a non-leader replica receives a client request, it forwards it to the last known leader, which ultimately provides a response to the client if valid. The leader adds the request as an entry to its log with the status *PROPOSE*, and subsequently sends the *Proposal* message as *PROPOSE (ballot, request)* to all the replicas. The proposed slot number in the ballot is incremented for each new request.
2) When a replica receives the proposal message, it verifies that it is from a known leader in the current term. It additionally checks for the validity or freshness of the request as per the ballot in the message. If all the checks pass, the replica will add the request to the log with status *PROPOSED* and provide a response to the leader in a *PROPOSEOK (ballot, commit_index)* message.
3) The leader requires a majority of *PROPOSEOK* messages to proceed with the commit and execute operations of the state machine. For each message received, the leader checks that the request has been previously logged with the *PROPOSE* status and continually checks for quorum. Once a quorum is reached, the leader replica updates its ballot, the operation number, and execution index. It then executes the operation and responds to the client. The log entry is also updated to reflect the *COMMITTED* and *EXECUTED* status, along with the result of the operation call. The leader sends the commit message (*COMMIT(ballot, commit_index)*) to all replicas. When a replica receives this message, it verifies the freshness of the ballot and confirms that the operation had been previously logged. The replica also has to ensure that all previous proposals have been committed requesting for state transfer where necessary. A *rebel* replica performs similar actions as the leader but does not provide a response to the client. The *witness* replica only logs the operation and sets its status to *Committed*.
4) The leader replica provides the result of the execution of the operation to the client. The client, upon receiving the response, updates the status of the corresponding request to prevent further redundant processing.

There are possible deviations from the normal processing of requests that can be encountered at any stage of the protocol. These issues and how LowPaxos can handle them to ensure a consistent state of the application are discussed in the next subsection (IV-B2).

2) RECOVERY IN LOWPAXOS

LowPaxos relies on the ballot, decision quorums, profiles and various attributes of a replica state (such as the replica's operation state when a request is received) for a consistent and high performance state machine. A replica operating in the normal mode will handle requests from known or higher terms, and strictly higher slot numbers. The replica's role at the time of request receipt dictates the subsequent course of action. If the term in the replica's ballot is lower than the one in the incoming proposal, it suggests that a new election has been successfully completed without the replica's awareness. Additionally, a new configuration has been generated and the replica will change its status to *CONFIGURATION* and request for this update from the leader. If the next slot number in the proposal ballot does not correspond to the replica's, then one of the replicas will require transfer of the log from the leader. A replica will change its status to *RECOVERY* and send the *REQUESTSTATE(ballot, commit_index)* message to the leader. The leader is also not immune to failure, for example in cases where it is isolated or marked offline by most replicas. The ballot information of subsequent interactions will dictate the operational status of a given replica and the next course of action. When a leader realizes that it is stale, it transitions its role to a rebel and mode to *CONFIGURATION* and requests for updated state information from the new leader.

The rebels will realize that the leader is unreachable if it does not respond to the periodic polls or send heartbeats while the witness only needs the heartbeat timeouts. A leader will be marked as unavailable and a replica will change its mode to *ELECTION*, and issue an offline election type request with

a new term. If the current leader receives the vote request, it will assert its leadership role by updating to a new term, creating a new configuration and sending it to the rest of the replicas.

a: PERFORMANCE DEGRADATION

The leader includes profile information in the poll responses and heartbeat messages sent to the replicas. If the performance of the leader degrades, then the replicas have to determine its degradation level in comparison with the recipient profile. A threshold is defined beyond which a replica can initiate another round of elections. The replica will change its mode to *ELECTION* and issue the *profile* election type request. When the leader receives this election request, it has to confirm that its performance has indeed degraded. If the performance of the leader is still acceptable, the leader will update its ballot term and create a new configuration to be sent to the replicas. If performance is confirmed as degraded, the leader changes its role to rebel and mode to *ELECTION* and checks the ballot of the replica seeking election. If the commit stage of this replica is behind, the leader (now rebel) will explicitly transfer the log and request it to restart the election type. Additionally, the leader will set a timer within which it will expect another election request or configuration. If there is a timeout, the leader will resume its role and continue processing client requests.

b: LOG TRANSFER

Replicas may fail or become unreachable and are either fully or partially involved in the state machine operations. At the next leader transaction (client request, poll or heartbeat), a replica may learn of this state. This requires that replicas request for new state or log data from the leader. The replica changes its mode to *RECOVERY* and issues the *REQUESTSTATE* message containing its current ballot and commit index. The additional request attributes ensure that the leader only provides a subset of the log to the replica. The leader confirms that the replica is outdated, retrieves the requested log portion and sends it to the requesting replica in form of a *LOGSTATE(ballot, commit_index, log)*. The log contains the data from the commit index of the request to the current commit index of the leader.

The recipient replica determines that the received state is valid (there is a part of it missing in its log). The replica processes the state information from its last commit index to that in the received state message according to its role. A rebel will commit and execute all the operations as it increments its operation number, the commit and execution indices. A witness only commits the received entries to its log. After all these recovery actions are completed, the replica will change its mode to *NORMAL*.

c: RECONFIGURATION

A number of role transitions are possible as a result of elections, request processing and other exchanges between replicas as shown in Figure 7. In a successful election, a configuration is generated to assign the rest of the replicas their respective roles. In the first election, the transition from a member to a role does not require additional state machine related operations. Subsequent configurations may require transitions, for example, a witness promoted to a rebel or a rebel demoted to a witness or a leader demoted to rebel/witness. Promotions require that the replica requests for a log portion from a leader if not updated with the latest information. Additionally, the new rebel will need to execute all operations committed if this had not been done before. The execution index is used to track the last log entry to have been executed by a replica. If a replica is demoted, the functions associated with the role at that instant shall proceed normally, with optional request for new state.

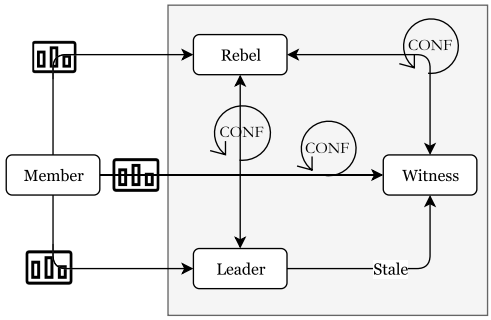

FIGURE 7. Role transitions in LowPaxos. A member will transition to leader, rebel or witness after an election. A leader transitions to a rebel or witness after it has been marked as offline by a majority of replicas and a new election has been successfully conducted. A witness can also be promoted to a rebel or a rebel is demoted to a witness after the leader reconfigures the setup based on new profile information. In most cases, these transitions necessitate state transfer from the current leader.

In a low resource setting, replicas may join or leave the system. It is imperative that the leader keeps track of the replicas to ensure that the configuration is updated. When a leader receives a message from a replica, its *alive* status is updated. The rebel status takes more precedence as this is a potential leader. If the leader doesn't hear from a rebel for an extended period of time, it will create a new configuration and broadcast it to the replicas. This necessitates a transition of one or more replicas from witness to rebel and subsequent log request and transfer. A new node joins the system by sending a *Join* message to one of the available replicas that will process (if leader) or forward it to the leader. The leader creates an updated configuration and shares with all the replicas. The role of the new replica will be witness but can transition to either leader or rebel at latter stages of the protocol.

C. LOWPAXOS GUARANTEES

LowPaxos provides the same guarantees as Paxos or MultiPaxos in both its leader election and consensus protocol components. Stateright [34], an actor-based model checker, is used to verify particularly the following properties:

- **Non-triviality**: Committed commands by the replica system must have been issued by a client in form of

Algorithm 1 LowPaxos Consensus Algorithm

Require: *request*, *requestID*, *ComIdx*, *ExecIdx*
Require: $r \in r_i, r_{i+1}, \ldots .r_n; n; b; t; l; p; s; d; o; R$ ▷ Replica r, n - number of replicas, b - ballot, t - election type, l - replica role, p - profile, s - status of replica, d - log or state at replica, o - operation number, R - leader replica
Require: θ ▷ Profile difference threshold

procedure HandleRequest(ballot, request)
  if $\neg R$ **then**
    StartElectionCycle(t)
  else
    if $r \leftarrow R$ & $s \leftarrow Normal$ **then**
      if *NewRequest* **then**
        $d \leftarrow request$
        Propose($b+1$, *request*)
      else
        ResendResponse
      end if
    else
      ForwardToLeader
    end if
  end if
end procedure
procedure HandlePropose(b, request)
  if $b \leftarrow Fresh$ **then**
    switch b **do**
      case *Next*
        $d \leftarrow request$
        ProposeOk(b)
      case *Future*
        ProposeOk(b)
        $s \leftarrow RECOVERY$
        RequestState(b, *ComIdx*)
  else*RequestConfig*(b)
  end if
end procedure
procedure HandleProposeOk(ballot)
  for $request \in ProposeOk(b)$ **do**
    if $request \in d$ **then**
      CheckQuorum
      if *Quorum* **then**
        Update(b, d, o, *ComIdx*)
        $result \leftarrow OperationCall$
        Response(*result*)
        Update(*ExecIdx*)
        Commit(b)
      end if
    end if
  end for
end procedure
procedure HandleCommit(b)
  if $b \leftarrow Next$ **then**
    switch l **do**
      case *Rebel*
        PerformOperation
        Update(b, d, *OpNumber*)
        Update(*ComIdx*, *ExecIdx*)
      case *Witness*
        Update(b, d, *OpNumber*)
        Update(*ComIdx*)
  end if
  if $b \leftarrow Future$ **then**
    $s \leftarrow RECOVERY$
    RequestState(*ComIdx*)
  end if
end procedure
procedure HandleStateRequest(b, idx)
  if $r \leftarrow R$ & $s \leftarrow Normal$ **then**
    if $b \leftarrow Old$ & $ComIdx > idx$ **then**
      $dp \leftarrow d(idx1, ComIdx)$
      SendState(*ComIdx*, dp)
    end if
  end if
end procedure
procedure HandleStateReceipt(idx, dp)
  for $entry \in dp$ **do**
    switch l **do**
      case *Rebel*
        PerformOperation
        Update(b, d(*entry*), *OpNumber*)
        Update(*ComIdx*, *ExecIdx*)
      case *Witness*
        Update(b, d(*entry*), *OpNumber*)
        Update(*ComIdx*)
  end for
  $s \leftarrow NORMAL$
end procedure
procedure processVote(j, b) ▷ Replica r receives vote from j
  for responseVote(b) **do**
    CheckQuorum
    if *Quorum* **then**
      $r \leftarrow$ 'R'
      $config \leftarrow CreateConfiguration$
    end if
  end for
end procedure
procedure processProfile(P_1, P_2)
  if $P_1 \geqslant P_2$ **then**
    true
  else
    false
  end if
end procedure

a request to one of the member replicas (leader, rebel or witness). In addition, the commands must have been proposed by a consensual leader.

```
97: procedure StartElectionCycle(t)
98:     procedure requestVote(i, b, l, P_{i,j})
99:         for j ∈ r do                ▷ When replicas receive the requestVote
100:            P_{j,i} ← Monitor(i)
101:            p ← processProfile(P_{j,i}, P_{i,j})
102:            switch t do
103:                case DEFAULT_or TIMEOUT
104:                    if p then
105:                        voted ← i, b, t
106:                        return responseVote(b)
107:                    else
108:                        l ← 'rebel'
109:                    end if
110:                case PROFILE_or OFFLINE
111:                    switch l do
112:                        case leader
113:                            if θ then
114:                                l ← 'rebel'
115:                                s ← 'ELECTION'
116:                                return responseVote(b)
117:                            else
118:                                assert(l ← R)
119:                            end if
120:                        case other
121:                            ProcessrequestasDefault
122:                case DEGRADED
123:                    return responseVote(b)
124:        end for
125:    end procedure
126: end procedure
```

- **Safety**: There is total ordering of the commands as issued by a client and proposed by the leader. Commands are identified by a tuple of client and request identifiers. In the proposal phase, an operation or proposal number is assigned to the command and advanced as part of the ballot for a consensus vote. All these identifiers are incremented monotonically to ensure progression.
- **Liveness**: A proposed command will eventually be committed by role-assigned members of the system as long as a majority of the replicas are available. Further, the commands are executed by the leader and rebel replicas and a response will be provided to the client. For a given term, a unique leader will ultimately be elected to propose commands.

V. EVALUATION

LowPaxos is evaluated against MPaxos and EPaxos as leader-based and leaderless protocols respectively. The evaluation will give a comparative idea of performance of the two types of protocols in a challenged environment. The evaluation sought to answer the following questions:

- How does LowPaxos compare with leader and leaderless protocols in performance when deployed in an environment with low and variable resources?
- What are the performance implications of changes in the properties of a challenged environment to a distributed application deployed in these settings?
- How does LowPaxos recover under extremities of challenged environments compared to MPaxos and EPaxos?

To answer these questions, two environments were used: A microbenchmark environment on CloudLab [32] and the production setting composed of nodes in different data center locations (cities) in Africa. Each environment consists of five nodes with the microbenchmark environment more uniformly distributed in computing and network resource setup.

A. EXPERIMENTAL SETTINGS

The production environment is mainly heterogeneous with variations in compute resources (Table 2) and network attributes. For example, the network latencies between nodes range from 5*ms* to 350*ms* at best. The nodes in the CloudLab environment are homogeneous 8-core 64GiB Memory HP ProLiant M510 Servers connected over a Local Area Network (LAN) via Mellanox ConnectX Interfaces. The nodes are installed with Ubuntu 18.04 LTS. The CloudLab environment allows for on-fly modification of node properties as experiments are run.

TABLE 2. Node Specifications for the Production Environment in Africa.

No.	Name	CPU (cores)	Memory (GiB)	Disk (GB)
1	Dar-es-Salaam	2	16	250
2	Lusaka	2	16	250
3	Kampala	2	16	250
4	Tororo	2	16	200
5	Johannesburg	16	64	1800

B. THROUGHPUT AND LATENCY

The MPaxos and EPaxos implementations used for evaluation are written in Go. In the initial experiment conducted within the controlled CloudLab environment, the aim was to compare the throughput and latency performance of the three protocols in a homogeneous setting, particularly with the inclusion of LowPaxos implemented in Rust. The network properties for each node interface were configured to operate at *1Gbps* with a latency of *20ms* and a packet loss of 0.0000001%. This is to introduce some arbitrary property values in order to maintain a level of homogeneity. Both the load generator (client) and replica nodes run with identical specifications, and the client issues closed-loop requests. The predetermined number of requests is sent, and measurements of latencies are recorded and subsequently used to calculate the throughput. Figure 8 illustrates comparative plots of throughput and total latency for a number of requests for the three protocols.

At lower request levels, the throughput of LowPaxos is initially lower but gradually catches up, ultimately converging with the other two protocols. During periods of low request volume, LowPaxos engages in leader election, and the initial requests may be directed to a non-leader node, thereby contributing to the lower throughput. However, as the load increases, the throughput stabilizes, facilitated by the establishment of a strong leader that allows the client to communicate directly, resulting in sustained competitive throughput. This observation implies that the Rust implementation of LowPaxos is not a significant influencing factor in the next experiments. Additionally, the total latency plot demonstrates that all three protocols operate within the same range in the homogeneous setting.

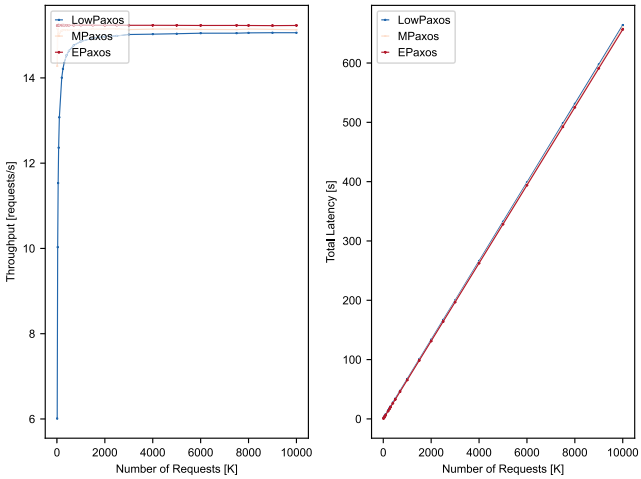

FIGURE 8. In homogeneous settings, LowPaxos, MPaxos, and EPaxos exhibit similar behaviors. During the initial election phase in LowPaxos, the determination of a strong leader by a client results in reduced throughput at lower request loads. As the request load increases, the throughput of LowPaxos competes favorably with the other two protocols. Notably, the total latencies eventually converge for all three protocols in this controlled setup.

In the production environment, the client is positioned within one of the five data centers. The protocols are initiated, and the client issues 10,000 requests to the replica system over a 220-second period. The results are depicted in Figure 9. The election of the most suitable leader in LowPaxos leads to lower request latencies, averaging 25*ms* in its optimal scenario, compared to EPaxos at 60*ms* and MPaxos at 135*ms*. This is mainly attributed to election of a strong leader (usually centralized), allowing consensus to be reached among most replicas in the shortest possible time. In MPaxos, the selection of the leader tasked with proposing commands may vary, and the most optimal choice is highly probabilistic. In EPaxos, commits on the fast-path can yield significant performance gains for non-conflicting requests. However, conflicting requests expose EPaxos to the leadership challenges of MPaxos.

C. RECOVERY

Recovery and adaptation to changing properties is an important design consideration of a consensus protocol for

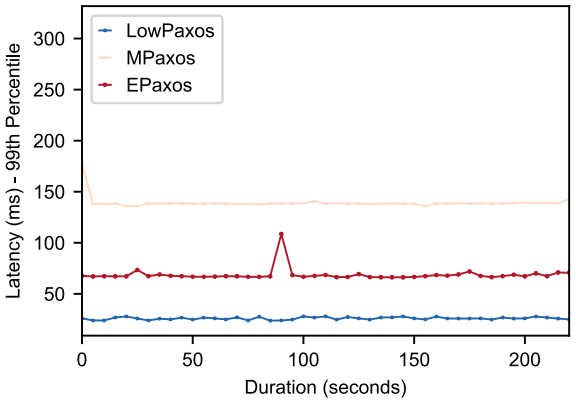

FIGURE 9. The 99th percentile latency plots for LowPaxos, MPaxos, and EPaxos (10% command conflict) in the production environment over a runtime duration of 220 *seconds* for 10,000 requests issued by a client in one of the data centers. The election of a strong leader can significantly improve the performance of a leader-based consensus protocol. In MPaxos, this is probabilistic and and inversely proportional to the number of replicas. EPaxos can perform better than MPaxos and LowPaxos for non-conflicting commands issued by clients to the immediate data center replica, but can suffer from MPaxos leadership challenges for conflicting requests.

challenged environments. In the CloudLab environment, the latency between the client and the replicas is set to 5*ms* and the network properties of the replicas varied (Table 3) as 7,500 client requests are issued. The network properties varied include the latency, packet loss and the link capacity. The requests are issued with the default network configuration and the link properties changed after every 120 *seconds* as shown in Figure 10. In LowPaxos, the changes lead to computation of new profiles as shown in Table 4, and in most cases will necessitate a profile election type if the current leader profile drops below a threshold (10% in the experimental setup). The network changes temporarily defaults the setup and hence the periodic lower request latencies for each of the protocols under consideration. In the default setup, the performance of the three protocols in the first 120 *seconds* is similar given the homogeneity of the environment, as shown in Figure 8. From $t = 125s$ to $t = 270s$, the degradation of the network setup introduces performance changes with most requests processed approximately within 220*ms* for all the three protocols. This is because of the minimal variation especially in the latency property of the three nodes required for consensus. From $t = 280s$ to $t = 415s$, further changes to the network setup degrades the performance with requests now processed approximately within 360*ms* for EPaxos and MPaxos while LowPaxos completes within 280*ms*. LowPaxos will choose a leader amongst replicas 2, 3 and 4 to achieve this lower processing latency. After $t = 415\ s$, the network setup is reset to the default and hence similar performance expected henceforth. The latency drops at $t = 120s$ intervals is due to the change script deleting the previous configuration before applying a new one. It should also be noted that LowPaxos has higher latency spikes

TABLE 3. Network property changes for the five replicas categorized into three, with Setup 0 as the default configuration. The changes are periodically made to the network interface using the Linux Traffic Control (tc) utility tool.

	Setup 0			Setup 1			Setup 2		
	T(Gbps)	L(ms)	J(10^{-6}%)	T(Kbps)	L(ms)	J(%)	T(Kbps)	L(ms)	J(%)
1	1	20	0.1	256	69	0.5	50	159	5
2	1	20	0.1	200	74	0.55	150	123	0.5
3	1	20	0.1	175	73	0.55	300	81	0.1
4	1	20	0.1	150	84	0.59	250	96	0.4
5	1	20	0.1	120	129	1	150	126	0.5

TABLE 4. Replica profiles (10^{-2}) for three network change setups in LowPaxos. The profile values are computed using Equation 1 and the network property values in Table 3. Other resource properties are kept constant for all the replicas in the evaluation.

	Setup 0					Setup 1					Setup 2				
Replica	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
1	1	0	0	0	0	1.00	0.97	1.00	0.92	0.79	1.00	0.14	0.16	0.15	0.13
2	0	1	0	0	0	0.86	1.00	0.90	0.78	0.68	0.31	1.00	0.41	0.37	0.33
3	0	0	1	0	0	0.86	0.80	1.00	0.76	0.65	0.77	0.84	1.00	1.00	0.83
4	0	0	0	1	0	0.72	0.71	0.70	1.00	0.58	0.45	0.50	0.69	1.00	0.49
5	0	0	0	0	1	0.45	0.44	0.44	0.43	1.00	0.30	0.33	0.40	0.37	1.00

especially at the onsets of leader election, reconfiguration and log recovery stages.

Further adaptation experiments are conducted in the heterogeneous environment consisting of the five replicas described in Table 2. These replicas run the three protocols, with client requests issued at intervals (the conflict rate for EPaxos client requests is set to 20%). LowPaxos, MPaxos, and EPaxos, when configured with optimal leaders, achieve similar throughput levels, with most requests completing within 100*ms*, as shown in Figure 11. However, the nature of a challenged environment demands dynamism in the operation of a consensus protocol. In LowPaxos, this dynamism is driven by changes in replica and network performance, often necessitating the election of a new leader. The new leader is at the apex of the performance pyramid and forms quorum with other replicas within the shortest period of time. The introduction of command conflicts, even at a low percentage, significantly degrades EPaxos's performance, with some requests taking approximately 575*ms* to complete. This delay occurs because requests must take the slow-path to ensure no other replica is proposing similar commands. In MPaxos, the long-term leader's performance initially shows better results but degrades over time. Since MPaxos is not adaptive, the protocol's performance is constrained by the leader's performance. The ability to elect a good leader and continually assess its strength in LowPaxos results in overall better performance compared to MPaxos and EPaxos. Although this process may require state transfer due to role changes, the protocol eventually stabilizes at better throughput levels compared to the other two protocols.

D. PERFORMANCE OVERHEAD

To analyze the performance overhead of LowPaxos relative to EPaxos and MPaxos, Prometheus [1] is deployed on a client node. Prometheus is a monitoring system that logs metrics in a time series database and offers a flexible query language PromQL. A local node exporter daemon tracks the CPU, memory, and network utilization of the replicas. Periodically, Prometheus scrapes these metrics from the replicas and stores them in its database. During each execution of the three protocols, the client generates 10,000 requests, and metric queries are collected throughout the entire request processing duration. The findings of the performance overhead evaluation are illustrated in Figure 12.

The CPU utilization of EPaxos is significantly higher (upto 4X) than that of LowPaxos and MPaxos. In EPaxos, each replica can lead the operations of the state machine. This requires that replicas keep additional state to advance the client requests. Each replica also tracks dependencies

[1] https://prometheus.io/

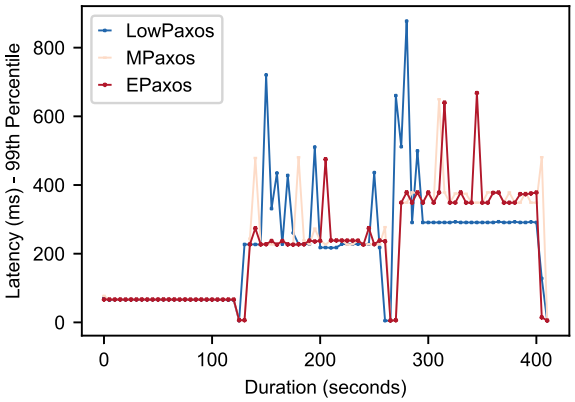

FIGURE 10. The 99th percentile latency plots for LowPaxos, MPaxos, and EPaxos (10% command conflict) in the CloudLab environment for 7,500 client requests. The homogeneity of the environment for duration $t = 0s$ to $t = 120s$ leads to the similar results as described in Figure 8. From $t = 12s$ to $t = 270s$, most of the protocols will complete request processing within 220*ms* due to a minimal variation in the latency property of the participating replicas. Additional changes to the network degrades the performance of MPaxos and EPaxos further, while LowPaxos election of a new strong leader ensures better comparative request latencies from $t = 280s$ to $t = 415s$.

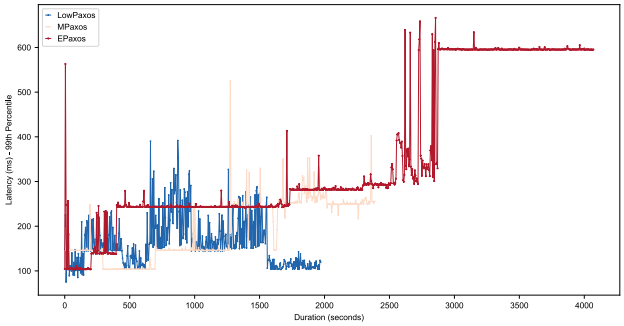

FIGURE 11. The 99th percentile latency plots for LowPaxos, MPaxos, and EPaxos (20% command conflict) in the heterogeneous challenged setting in Africa for 16K client requests. With optimal leaders, LowPaxos, MPaxos and EPaxos achieve similar throughput levels as most requests are processed within 100*ms*. As the challenged environment is susceptible to change, the adaptive nature of LowPaxos ensures that a new leader is elected (and high performance configuration of replicas) when a performance threshold is reached. This ensures that LowPaxos has a better overall throughput in comparison to MPaxos and EPaxos.

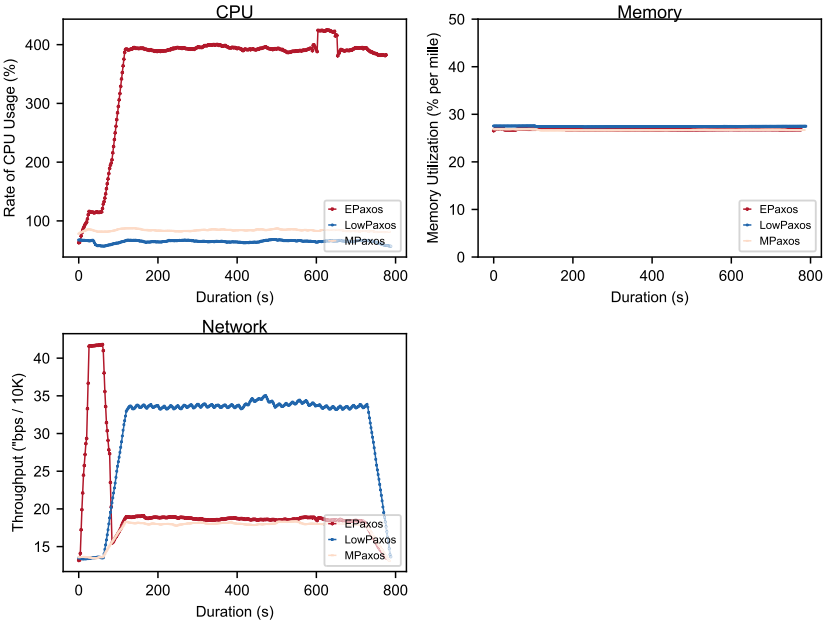

FIGURE 12. The peformance overhead of LowPaxos, EPaxos and MPaxos for 10K requests on the microbenchmark environment. The CPU, Memory and Network utilization are tracked for the duration of each protocol execution. The results show that CPU utilization in EPaxos is upto 4X of LowPaxos and MPaxos. The memory utilization for both protocols is largely similar. The network overhead of LowPaxos is higher compared to the other two protocols as extra information is required to relay topology changes.

between proposed values to ensure consistency and correctness. These are CPU-intensive operations and explains the high CPU load in EPaxos. LowPaxos and MPaxos rely on a single leader and hence the CPU load concentration is on this replica. However, the overall CPU utilization is lower for the two protocols compared to EPaxos. For the duration of the execution, the memory utilization of the three protocols is largely similar.

The network throughput for EPaxos at the start of the execution is high but quickly stabilizes to lower rates. This is partly down to initial exchanges between replicas for dependencies and possible conflicts and their resolution. Given that most of the requests are non-conflicting, additional network overhead is not required as the fast path will be used for commits. The use of the long-term leader in MPaxos keeps the network utilization relatively uniform for the entire duration of the execution. In LowPaxos, the higher network overhead is attributed to monitoring and liveness checks. For example, there are extra messages required to periodically share profile information. In addition, the leader notifies all replicas of its status while the rebels keep probing the leader. Despite this overhead, the performance of LowPaxos matches EPaxos and MPaxos in a homogenous setting as shown in Figure 12. Efforts to minimize the network overhead shall be explored in future work.

VI. CONCLUSION AND FUTURE WORK

The components constituting a distributed system are prone to failure during any phase of the system's operation. Failures can arise from issues related to communication and performance within the distributed elements. These challenges add complexity to the design and functioning of resilient and highly available distributed systems, which have become increasingly necessary for ensuring quality of service. On one hand, having greater control over the environment can offer operational guarantees, facilitating the attainment of desirable performance levels. Conversely, environments facing challenges contend with heightened uncertainties, including limitations and fluctuations in resources that hinder the optimal operation of the system. This further complicates the design and implementation of distributed systems within such settings.

This paper presents LowPaxos as a distributed consensus protocol designed specifically for challenging environments. LowPaxos leverages the unique properties of these settings to identify the long-term leader, determining its relative strength compared to the other replica members. The protocol integrates the consideration of resource variations, enabling it to adapt to the typical changes in properties of challenged environments, where deviations from the norm are more common. The experimental findings show that incorporating these attributes into the leader election process yields better performance gains when compared to both leader-centric and leaderless protocols. Moreover, LowPaxos demonstrates better adaptability to the dynamic and evolving changes that are characteristic of low-resource settings, showcasing its relevance in challenged and resource-constrained environments.

The extension of this research will involve implementing further optimizations to enhance LowPaxos, including the incorporation of command batching to achieve improved

performance. Numerous consensus protocols have illustrated that batching can yield significant performance gains, with some studies reporting enhancements of up to 4.5 times [33]. Another extension aspect will be on snapshot management, to ensure control over log sizes. As more data is accumulated, integration of machine learning techniques such as parallel deep neural networks [39] to manage role designations shall be explored.

ACKNOWLEDGMENT

The authors would like to thank all contributions in feedback and resources towards this research publication. A special thanks to the University of Washington and the Paul Maritz Foundation for the fellowship, and colleagues at the Systems Laboratory for collaboration efforts especially Priyal Suneja, Kevin Zhao, and Henry Schuh. They also thank Cloudlab [32] and UbuntuNet Alliance [2] for the evaluation infrastructure, the Platform and Testbed for Improved Observability of African Networks (PATIO) project and Government of Uganda through Makerere University Research and Innovation Fund (RIF).

REFERENCES

[1] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst. (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.
[2] *ETCD*. Accessed: Oct. 26, 2023. [Online]. Available: https://etcd.io/
[3] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proc. 7th Symp. Operating Syst. Design Implement.*, 2006, pp. 335–350.
[4] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for Internet-scale systems," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2010, p. 11.
[5] J. L. Carlson, *Redis in Action*. New York, NY, USA: Manning, 2013.
[6] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, pp. 56–78, Feb. 1991.
[7] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, Nov. 2013, pp. 358–372.
[8] S. Tollman, S. J. Park, and J. Ousterhout, "EPaxos revisited," in *Proc. 18th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2021, pp. 613–632.
[9] L. Lamport, "Paxos made simple," *ACM SIGACT News, Distrib. Comput. Column*, vol. 32, no. 4, pp. 51–58, Dec. 2001.
[10] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 305–319.
[11] C. S. Barcelona, "Mencius: Building efficient replicated state machines for WANs," in *Proc. 8th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2008, pp. 369–384.
[12] K. Ngo, S. Sen, and W. Lloyd, "Tolerating slowdowns in replicated state machines using copilots," in *Proc. 14th USENIX Symp. Operating Syst. Design Implement.*, 2020, pp. 583–598.
[13] H. Howard, D. Malkhi, and A. Spiegelman, "Flexible paxos: Quorum intersection revisited," 2016, *arXiv:1608.06696*.
[14] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar, "WPaxos: Wide area network flexible consensus," 2017, *arXiv:1703.08905*.
[15] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy, "Designing distributed systems using approximate synchrony in data center networks," in *Proc. 12th USENIX Symp. Networked Syst. Design Implement. (NSDI)*, 2015, pp. 43–57.
[16] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, "Just say NO to Paxos overhead: Replacing consensus with network ordering," in *Proc. 12th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2016, pp. 467–483.
[17] J. Guarnieri and A. Charapko, "Linearizable low-latency reads at the edge," in *Proc. 10th Workshop Princ. Pract. Consistency Distrib. Data*, May 2023, pp. 77–83.
[18] T. Nyirenda-Jere and T. Biru, "Internet development and Internet governance in Africa," *Internet Soc.*, pp. 1–44, May 2015. [Online]. Available: https://www.internetsociety.org/wp-content/uploads/2017/08/InternetInAfrica-2015070820Final.pdf
[19] B. Charyyev, E. Arslan, and M. H. Gunes, "Latency comparison of cloud datacenters and edge servers," in *Proc. IEEE Global Commun. Conf.*, Dec. 2020, pp. 1–6.
[20] O. Victor Babasanmi and J. Chavula, "Measuring cloud latency in Africa," in *Proc. IEEE 11th Int. Conf. Cloud Netw. (CloudNet)*, Nov. 2022, pp. 61–66.
[21] T. Biswas, R. Bhardwaj, A. K. Ray, and P. Kuila, "A novel leader election algorithm based on resources for ring networks," *Int. J. Commun. Syst.*, vol. 31, no. 10, p. e3583, Jul. 2018.
[22] G. Ishigaki, R. Gour, A. Yousefpour, N. Shinomiya, and J. P. Jue, "Cluster leader election problem for distributed controller placement in SDN," in *Proc. IEEE Global Commun. Conf.*, Singapore, Dec. 2017, pp. 1–6, doi: 10.1109/GLOCOM.2017.8254748.
[23] V. Karpov and I. Karpova, "Leader election algorithms for static swarms," *Biologically Inspired Cogn. Archit.*, vol. 12, pp. 54–64, Apr. 2015.
[24] Y. Zuo, W. Yao, Q. Chang, X. Zhu, J. Gui, and J. Qin, "Voting-based scheme for leader election in lead-follow UAV swarm with constrained communication," *Electronics*, vol. 11, no. 14, p. 2143, Jul. 2022, doi: 10.3390/electronics11142143.
[25] B. Awerbuch, "Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems," in *Proc. 19th Annu. ACM Conf. Theory Comput. (STOC)*, 1987, pp. 230–240.
[26] Q. Dong and D. Liu, "Resilient cluster leader election for wireless sensor networks," in *Proc. 6th Annu. IEEE Commun. Soc. Conf. Sensor, Mesh Ad Hoc Commun. Netw.*, Rome, Italy, Jun. 2009, pp. 1–9, doi: 10.1109/SAHCN.2009.5168966.
[27] V. Raychoudhury, J. Cao, and W. Wu, "Top K-leader election in wireless ad hoc networks," in *Proc. 17th Int. Conf. Comput. Commun. Netw.*, St. Thomas, VI, USA, Aug. 2008, pp. 1–6, doi: 10.1109/icccn.2008.ecp.35.
[28] J. Maeng and I. Joe, "Energy-based leader election (E-LE) for group management of IoT," in *Software Engineering Perspectives in Systems* (Lecture Notes in Networks and Systems), R. Silhavy, Ed., Cham, Switzerland: Springer, 2022, pp. 177–184, doi: 10.1007/978-3-031-09070-7_15.
[29] X. Xu, L. Hou, Y. Li, and Y. Geng, "Weighted RAFT: An improved blockchain consensus mechanism for Internet of Things application," in *Proc. 7th Int. Conf. Comput. Commun. (ICCC)*, Chengdu, China, Dec. 2021, pp. 1520–1525, doi: 10.1109/ICCC54389.2021.9674683.
[30] D. Pearce, J. Dunlop, and R. C. Atkinson, "Leader election in a personal distributed environment," in *Proc. IEEE 16th Int. Symp. Pers., Indoor Mobile Radio Commun.*, Berlin, Germany, Oct. 2005, pp. 1307–1311, doi: 10.1109/PIMRC.2005.1651652.
[31] L. Zamir, A. Shaan, and M. Nojoumian, "ISRaft consensus algorithm for autonomous units," in *Proc. IEEE 29th Int. Conf. Netw. Protocols (ICNP)*, Dallas, TX, USA, Nov. 2021, pp. 1–6, doi: 10.1109/ICNP52444.2021.9651979.
[32] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Jul. 2019, pp. 1–14.
[33] M. Whittaker, N. Giridharan, A. Szekeres, J. M. Hellerstein, and I. Stoica, "Compartmentalized consensus: Agreeing with high throughput," Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, CA, USA, 2020.
[34] *Stateright*. Accessed: Nov. 25, 2023. [Online]. Available: https://www.stateright.rs
[35] V. Arora, T. Mittal, D. Agrawal, A. El Abbadi, and X. Xue, "Leader or majority: Why have one when you can have both? Improving read scalability in raft-like consensus protocols," in *Proc. 9th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2017, p. 14.
[36] L. Lamport, "The part-time parliament," in *ACM Trans. Comput. Syst.*, vol. 16, no. 998, pp. 133–169, 2019.
[37] B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proc. 7th Annu. ACM Symp. Principles Distrib. Comput.*, 1988, pp. 8–17.

[2] https://ubuntunet.net/

[38] B. Liskov and J. Cowling, "Viewstamped replication revisited," MIT, Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2012-021, 2012.
[39] S. Khan, M. A. Khan, M. Khan, N. Iqbal, S. A. AlQahtani, M. S. Al-Rakhami, and D. M. Khan, "Optimized feature learning for anti-inflammatory peptide prediction using parallel distributed computing," *Appl. Sci.*, vol. 13, no. 12, p. 7059, Jun. 2023. [Online]. Available: https://www.mdpi.com/2076-3417/13/12/7059

ALEX MWOTIL received the B.Sc. and M.Sc. degrees in computer science from Makerere University, Kampala, where he is currently pursuing the Ph.D. degree. He is the product lead at Crane Cloud an abstraction platform for deployment and management of containerized applications in resource constrained settings. Previously, he has worked as a Systems Engineer with different research and academic institutions in Uganda. His research interests include cloud for resource-constrained settings and trust and identity services in Africa.

THOMAS ANDERSON is currently a Professor with the Paul G. Allen School of Computer Science and Engineering, University of Washington. His research interests include building practical, robust, and efficient computer systems, including distributed systems, operating systems, computer networks, multiprocessors, and security. He is the winner of the USENIX Lifetime Achievement Award, the USENIX STUG Award, the IEEE Koji Kobayashi Computer and Communications Award, the ACM SIGOPS Mark Weiser Award, and the IEEE Communications Society William R. Bennett Prize.

BENJAMIN KANAGWA is currently an Associate Professor of software engineering with Makerere University, Kampala, and heads the software systems center whose main focus is to develop collaborations with industry leading to high end technology startups and adoption. His research revolves around software engineering with a special emphasis on software architectures, especially service oriented systems, microservice, and cloud computing. He is also involved in open source initiatives, including Helecare2x, a solution aimed at helping health facilities automate patient records.

THEANO STAVRINOS received the Ph.D. degree from Princeton University, in May 2023. She is currently a Postdoctoral Researcher with the University of Washington working on the Treehouse and Future of Cloud Infrastructure (FOCI) projects. She is exploring how to build storage systems that will support a power control plane for cloud systems. Her research interests include distributed systems, caching, storage, and systems sustainability.

ENGINEER BAINOMUGISHA is currently an Associate Professor of computer science and the Chair of the Department of Computer Science, Makerere University. His research focuses on computer science-driven solutions to the prevailing world challenges. He leads several innovative and research initiatives that aim to create and apply computational methods and tools that can improve the quality of life, especially in the developing world setting. His current research interests include building and deploying distributed systems of miniature the IoT/mobile devices to measure and derive trends of air quality in major cities in Africa.