Leader Set Selection for Low-Latency Geo-Replicated State Machine

Shengyun Liu and Marko Vukolić

Abstract—Modern planetary scale distributed systems largely rely on a State Machine Replication protocol to keep their service reliable, yet it comes with a specific challenge: latency, bounded by the speed of light. In particular, clients of a *single-leader* protocol, such as Paxos, must communicate with the leader which must in turn communicate with other replicas: inappropriate selection of a leader may result in unnecessary round-trips across the globe. To cope with this limitation, several *all-leader* and *leaderless* alternatives have been proposed recently. Unfortunately, none of them fits all circumstances. In this article we argue that the "right" choice of the number of leaders depends on a given replica configuration and the workload. Then we present Droopy and Dripple, two sister approaches built upon state machine replication protocols. Droopy dynamically reconfigures the set of leaders. Whereas, Dripple coordinates state partitions wisely, so that each partition can be reconfigured (by Droopy) separately. Our experimental evaluation on Amazon EC2 shows that, Droopy and Dripple reduce latency under imbalanced or localized workloads, compared to their native protocol. When most requests are non-commutative, our approaches do not affect the performance of their native protocol and both outperform a state-of-the-art leaderless protocol.

Index Terms—State machine replication, geo-replication, latency optimization, state-partitioning.

1 Introduction

Modern internet applications [2], [3], [4] make use of State Machine Replication (SMR) protocols such as Paxos [5], [6], as a basic synchronization primitive to provide reliable service. Replication however becomes challenging at the planetary scale due to latencies that cannot be masked, being bounded by the speed of light. In this case we talk about *geo-replication*, with replicas and clients scattered across multiple remote *sites*.

To illustrate the problem, consider classical SMR protocols such as Paxos that have a *single leader* responsible for sequencing and proposing clients' requests. These proposed requests are then replicated across a majority of replicas, executed in order of their logical sequence numbers, with application-level replies eventually sent to the clients. For clients that are remote from the leader, this may imply costly round-trips across the globe.

Many latency-optimized SMR protocols tailored to geo-replication have been proposed recently [7], [8], [9], [10], [11]. These protocols (and Paxos) are similar in that they rely on a (variant of) consensus primitive which ensures resilience despite failures. Their differences have to do with the way they ensure linearizability [12] or strict serializability [13], i.e., strong consistency among all clients' requests. In general, these protocols can be further classified into two categories: *all-leader* protocols and *leaderless* protocols.

In *all-leader* protocols, the total order is pre-established based on, e.g., logical sequence numbers (e.g., Mencius [10]) or physical clocks (e.g., Clock-RSM [11]). A request can be proposed and sequenced by *any* replica, where every replica can act as a leader, typically by partitioning the sequence number space. In these protocols, a client submits its requests to a nearby replica, avoiding the communication with a single (and possibly remote) leader. Just like classical single-leader Paxos, all-leader SMR protocols work with a conservative assumption that clients' requests do not commute — hence all-leader protocols still require replica coordination to maintain a total order across all requests.

A challenge for all-leader SMR protocols is the coordination with distant or slow replicas, which can be the bottleneck, or even block the progress entirely. In some sense, the performance is determined by the "slowest" replica: this causes what is known as a "delayed commit" problem [10]. Roughly speaking, the delayed commit problem (that we detail in Section 2.2) arises from the need to confirm that all requests with an earlier sequence number are not "missed". For the most typical, imbalanced workloads, e.g., if most requests originate from clients that gravitate to a given site S, this incurs communication with *all* replicas including remote and slow ones. In this case, all-leader SMR may have worse performance than single-leader SMR, in which replication involves only S and the majority of sites closest to S.

On the other hand, *leaderless* protocols (e.g., Generalized Paxos [7] or EPaxos [9]) exploit the possible *commutativity* of requests and execute such requests (e.g., requests accessing distinct parts of state) out of order at different replicas. In a leaderless protocol, a request is typically proposed directly by a client or by any replica, and committed with a round-trip latency to replicas belonging to a *fast-quorum*. [1] Every

- *S. Liu is with College of Computer Science, National University of Defense Technology (NUDT), Changsha, China, 410073. Work done while being a PhD student at EURECOM. E-mail: liushengyun@nudt.edu.cn, lius@eurecom.fr*
- *M. Vukolić is with the IBM research - Zurich, Rueschlikon, Switzerland, 8803. E-mail: mvu@zurich.ibm.com*
- *Preliminary version of this work was presented in [1].*

1. A fast-quorum is larger than a majority [14] - we postpone a more detailed discussion to Sec. 2.3.

replica in a fast-quorum checks potential conflict among concurrent requests. Unfortunately, if (1) a conflict is detected, *or* (2) available replicas are fewer than a fast-quorum, one or more additional round-trip message exchanges to a majority of replicas are introduced in order to resolve the possible conflict. Note that leaderless protocols do not suffer from the delayed commit problem since no order is predefined.

In summary, there is no method that fits all situations and the choice of the "best" protocol largely depends on the specific deployment configuration and the workload. Namely, existing protocols are based on one of the following assumptions: (1) requests come mostly from within the vicinity of a single site (favoring classical single-leader SMR); (2) requests are evenly distributed among all sites and most concurrent requests do not commute (favoring all-leader SMR); or, (3) most concurrent requests commute (favoring leaderless SMR). More than often, none of these assumptions is always true in practice. For instance, due to time zone differences, clients of modern applications (such as online shopping and social networking) located at a given site may have different access patterns at different times of a day, dynamically changing the "popularity" of sites and possibly also the workload balance [15].

In this article, we present Droopy and Dripple, two sister approaches that explore the ways to mitigate issues of all-leader and leaderless SMR. The design considerations behind Droopy and Dripple are straightforward:

1) removing unnecessary dependencies to avoid delayed commit; and,
2) pre-grouping non-commutative requests to avoid inefficient conflict resolution.

Based on these two guidelines, Droopy is designed to dynamically reconfigure the leader set which can contain anything from a single to all replicas. The selection of leaders is based on previous workload and network condition. Dripple in turn is a state partitioning and coordination algorithm inspired by leaderless protocols. Dripple allows Droopy to reconfigure the leader set of each partition individually, at the same time ensuring to provide strong consistency, i.e., strict serializability [13] as a whole.

Although Droopy and Dripple can be applied to any leader-based SMR protocol, we implement Droopy and Dripple on top of a state-of-the-art all-leader SMR protocol — Clock-RSM [11] (we call our variant D^{2}Clock-RSM). On Amazon EC2 platform we evaluate D^{2}Clock-RSM under imbalanced workloads and three representative types of balanced workloads. For a more comprehensive evaluation, we also implement several state-of-the-art SMR protocols, i.e., Paxos, native Clock-RSM, Mencius and EPaxos [9] by making use of the same code base. Our experimental evaluation shows that, under typical imbalanced workloads, Droopy enabled Clock-RSM effectively reduces latency compared to its native protocol. Besides, under balanced but request-commutative workloads, where requests issued by each site access distinct part of state, D^{2}Clock-RSM outperforms native Clock-RSM, Paxos and Mencius, and achieves the performance similar to that of EPaxos — a state-of-the-art leaderless protocol. In contrast, under balanced but request-non-commutative workloads, D^{2}Clock-RSM's latency is similar to that of native Clock-RSM, and both achieve lower latency than EPaxos.

The rest of this article is organized as follows. In Sec. 2 we discuss, in the context of related work, how delayed commit problem (in all-leader SMR) and non-commutative requests (in leaderless SMR) affect latency. In Sec. 3 we describe the system model and assumptions. In Sec. 4 we give the overview and in Sec. 5 the details of Droopy and Dripple. Sec. 6 evaluates our approaches under several workloads. Sec. 7 concludes the article.

2 Related work

Since Lamports' seminal Paxos protocol [5], [6], numerous variants have been proposed [7], [8], [9], [10], [11], [16], [17], [18] in order to mitigate the latency problem for geo-replicated state machine. In general, these protocols fall into two categories: *all-leader* protocols, that allow every replica to act as a leader and propose requests piggybacked with a totally ordered sequence index; and *leaderless* protocols, that allow every client or replica to propose requests without pre-imposing total order, but detect conflicts dynamically.

2.1 Paxos overview

Paxos replicates requests (and application state) across $2t+1$ replicas, out of which t replicas can crash. In *common case* (see message pattern in Fig. 1), where there is no crash fault or network asynchrony, a distinguished replica called the *leader*, proposes requests by assigning each request a logical *sequence number*. The property that every (correct) replica agrees on the same request despite failures is ensured by a consensus sub-protocol called Synod [5], which lies at the heart of Paxos. In each consensus instance, the leader proposes a request by propagating the request to all replicas, waits until a majority of replicas have replicated the request, and finally decides the request. To maintain a total order among all requests, Paxos proposes each request at a separate consensus instance, which in turns has a unique logical sequence number, based on which all consensus instances are ordered.

In case the leader crashes or be partitioned from others, a new leader is elected and the most recent state (i.e., all decided requests) must be retrieved before processing new requests.

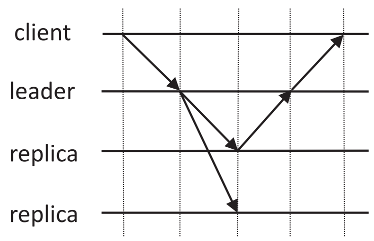


Fig. 1: The message pattern of Paxos (Phase 2) in common case when $t = 1$.

2.2 All-leader protocols (delayed commit)

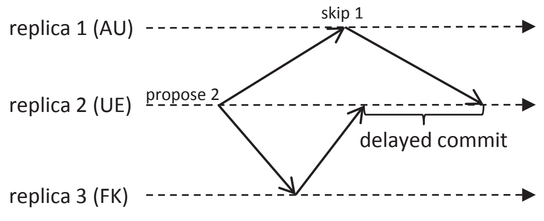
In order to optimize perceived latency for clients scattered across the planet, every replica in all-leader SMR protocols can act as a leader and assign a pre-ordered *sequence index* (e.g., a logical sequence number or a physical timestamp) to a request.

Mencius [10] facilitates multiple leaders by evenly pre-partitioning sequence number space across *all* replicas (modulo number of replicas), so that each replica can propose requests. For example, with 3 replicas, replica 1 sequences requests at sequence numbers 1,4,7..., replica 2 sequences requests at 2,5,8,..., and replica 3 sequences requests at 3,6,9,.... To mitigate execution delays, if a replica lags behind, it skips some sequence numbers by assigning *no-op* requests (or *empty* requests) to skipped sequence numbers, in order to catch up with other replicas. Such a *skipping* replica must however let its peers know which sequence numbers it skips — leading to what is known as the "delayed commit" problem. Intuitively, the delayed commit problem arises when the workloads or the latencies across replicas are non-uniform, which is most often actually the case.

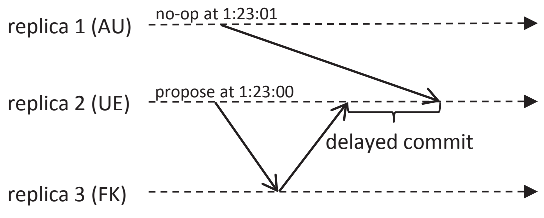
We illustrate the delayed commit problem in Mencius by an example with $n = 3$ replicas deployed on Amazon EC2. The round-trip latencies among 6 sites on Amazon EC2 is shown in Fig. 5 (Sec. 6). The example is shown in Fig. 2a. Assume that replica at UE (replica 2 in Fig. 2a) proposes request R_1 at sequence number 2, whereas replica at AU (replica 1 in Fig. 2a) is responsible for proposing request at sequence number 1. Because of the imbalanced workload, replica AU has not proposed any request when it receives proposal of R_1 from replica UE, at which time replica AU skips sequence number 1. Only upon replica UE receives the skip message for sequence number 1, it can commit and execute R_1 locally. Hence, a round-trip latency from UE to AU is introduced (231 ms), which is much larger than a round-trip latency from replica 2 to a majority (including IR besides UE itself) that a solution based on a single UE leader would require (88 ms).

When confronted with a slow replica, Fast Mencius [19] proposes two mechanisms named "Active Revoke" and "Multi-instance Propose" to standard Mencius, in order to speed up the advancement of normal replicas. Active Revoke actively proposes a no-op request at instances which should be proposed by a slow replica. Whereas, Multi-instance Propose allows a slow replica to finally commit its requests even with Active Revoke. Active Revoke and Multi-instance Propose do not target imbalanced workloads or non-uniform network latencies such as the example given above, and are orthogonal to our approaches introduced in this article.

More recently, Clock-RSM [11] was proposed with the goal of mitigating the delayed commit problem of Mencius by using loosely synchronized clocks. In Clock-RSM, each replica proposes requests piggybacked with its physical clocks, which is used instead of logical sequence numbers to order requests. In a similar way to Mencius, before executing a request, each replica is obliged to confirm that all previous requests have been executed locally. This requirement implicitly implies that no request with an earlier timestamp will be proposed later by any replica. In order to achieve this requirement efficiently, especially under imbalanced workloads, replicas in Clock-RSM exchange their physical clocks periodically (e.g., every 5 ms in its implementation) to



(a) Mencius.



(b) Clock-RSM.

Fig. 2: Delayed commit problem in state-of-the-art all-leader protocols.

notify other replicas that requests with an earlier timestamp have been sent already. Note that this is guaranteed by assuming FIFO channel between replicas. Whenever clocks at different replicas are synchronized, Clock-RSM reduces one-way latency for the delayed commit problem in Mencius. That is to say, each replica frequently notifies others that "no more request" will be proposed before a given timestamp.

However, delayed commit still exists in Clock-RSM, albeit being less pronounced. In example in Fig. 2b, just after replica UE proposes R_1, replica AU sends its current clock time to all replicas. Upon UE receives the clock time from AU, it confirms that no request with an earlier timestamp can be proposed by replica AU. Nevertheless, one-way latency from AU to UE is still larger than the round-trip latency from UE to a majority.

Now we formally define the delayed commit problem.

Definition 1. (***Delayed commit***) *If (i) replica s_2 is proposing request req_2 at sequence index sn_2, and (ii) replica s_1 is responsible for proposing request at $sn_1 < sn_2$, then s_1 may delay the commitment of req_2 at s_2, if s_1 takes longer time than a round-trip latency from s_2 to a majority to either (1) notify s_2 that no request will be proposed at sn_1, or (2) make request at sn_1 committed at s_2.*

To sum up, all-leader SMR protocols implicitly assume that every replica proposes requests at approximately the same rate and concurrent requests are non-commutative. Hence, these protocols allow every replica to act as a leader and are forced to maintain a global order among *all* requests. However, due to heterogeneity of wide area network and imbalance of workloads, all-leader protocols can suffer from the delayed commit problem.

2.3 Leaderless protocols (with conflict resolution)

Unlike single-leader Paxos or its all-leader variants, leaderless protocols [8], [9], [16] reduce latency by sequencing requests in a somewhat more decentralized and less ordered fashion. Namely, in leaderless protocols, each request is directly proposed by the client or by any replica. A proposed request is first sent to all replicas. Upon receiving the request, each replica individually and tentatively assigns a sequence number to the request, possibly adding the information about other concurrent requests, and replies to the sender. In case a *fast-quorum* of replicas have given a request the same sequence number, the request is committed. Otherwise, another one or more round-trip message exchanges are introduced in order to arbitrate conflicts. This design is motivated by an aggressive assumption that most concurrent requests are commutative (e.g., more than 98% [9]).

The number of replicas in a fast-quorum is larger than a majority [14]. For instance, it was proved [16] that, for a system using the minimal number of $n = 2t + 1$ replicas, if a request is committed after a round-trip communication directly from the client to a fast-quorum (which yields only two hops in solving consensus, and hence, SMR), then the number of replicas in a fast-quorum is at least $\lceil \frac{3}{2}t \rceil + 1$.

Fast Paxos [16] allows its clients to send requests directly to all replicas and matches the $\lceil \frac{3}{2}t \rceil + 1$ fast-quorum lower bound. In case a collision is detected, i.e., there are not enough replicas giving the same sequence number to a request, Fast Paxos relies on a single leader to re-orders the request.

Generalized Paxos [7] further reduces latency of Fast Paxos by exploiting *commutativity* among concurrent requests. In case two requests commute (e.g., request 1 writes object A, request 2 writes object B), these two requests can be executed out of order at different replicas. Generalized Paxos still relies on a single leader sub-protocol to resolve the order of non-commutative requests.

EPaxos [9] reduces the number of replicas in the fast-quorum by exactly one replica compared to Generalized Paxos. Note that EPaxos can achieve this since it introduces one more communication step (i.e., the third hop) from the client to a nearby replica, say command leader, at the first step (hence the $\lceil \frac{3}{2}t \rceil + 1$ fast-quorum lower bound does not apply in this case). The command leader then binds a sequence number (which is not totally ordered among replicas) and a set of conflicting (non-commutative) requests known by the command leader to the request. Upon receiving a proposal from any command leader, each replica updates the conflict set of the received request and re-calculates the sequence number based on the new dependency information, then replies to the command leader. Upon the command leader receives the same sequence number and conflict set from a fast quorum, the request is committed and can be executed orderly based on dependency information. Otherwise, the command leader relies on another round-trip communication among a majority to confirm the order eventually.

In summary, existing leaderless protocols are designed based on an aggressive assumption that most concurrent requests are commutative. The performance of leaderless protocols is hence driven by the number of non-commutative requests as well as the size of fast-quorum. For instance, in a system with 5 replicas in Fig. 5, if replica at UE proposes a request which is conflicting with a concurrent request proposed by JP (Japan), then at least another round-trip latency among a majority (80ms and 107ms, respectively) is introduced.

3 System model

We discuss a distributed system which contains a set Π of $n = |\Pi|$ *machines*, also called *processes* or *replicas*. More specifically, we target the State Machine Replication (SMR) problem [20] that ensures *strong consistency*, i.e., *linearizability* [12] or *strict serializability* [13], of a replicated and deterministic service.

We assume there are a total of $n \geq 2t + 1$ replicas (or sites), $s_1, s_2, ..., s_n$, among which t can crash (but not behave arbitrarily). We also assume there are an unbounded number of clients which can issue requests and fail by crashing.

Network. We allow for asynchrony in that communication time between any two replicas is not bounded — however, to circumvent the FLP impossibility [21], we assume the system to be eventually synchronous. We further assume a FIFO channel between any two replicas, i.e., messages from one replica to another are delivered by the destination in order. For simplicity reason, we also assume that the latency of every communication channel is symmetric, which implies that one-way latency from one replica to another equals the value of the opposite direction[2].

Clocks. When referring to clock-based systems (e.g., Clock-RSM), we assume there is a physical clock equipped at each replica. Clocks at different replicas are loosely synchronized using a clock synchronization protocol such as Network Time Protocol (NTP) [22].

Terminologies. Following the terminology of existing SMR protocols, we say replica s_i *proposes* a request req at sequence index sn (e.g., sequence number or physical clock) if s_i is the leader for sn and s_i sends a PROPOSE message which contains req and sn to all replicas. Similarly, we say s_i *commits* req at sn if s_i confirms that a majority of replicas have replicated req at sn and all requests with a smaller sequence index have been committed by s_i.

Linearizability versus (strict) serializability. Droopy is designed upon single-leader or all-leader protocols, so it provides the same consistency guarantee as that of native protocols, i.e., linearizability. Linearizability abstracts the system state as an indivisible object or register, which is accessed (i.e., read or write) by every request.

In contrast, Dripple explores request commutativity by dividing the system state into m disjoint partitions $p_1, p_2, .., p_m$ by, e.g., range partitioning. A partition can be a single object or a group of objects . Each request can access several objects across several different partitions, specified at the time when the request is received by any replica. Therefore, we further assume that, for each request, applications using Dripple can provide the information regarding which part of state is involved, i.e., which partitions that the request will potentially access.

By leveraging state-partitioning, Dripple instead ensures strict serializability, which is widely adopted in transactional semantics such as in database systems. In general, strict serializability ensures that the execution order among all requests is equivalent to a sequential one, and this sequential execution respects real-time order (i.e., wall-clock). Real-time order implies that, if request R_1 is executed before

2. Measuring one-way latency in WAN is usually non-trivial.

another request R_2 is issued, then the sequential execution should reflects that R_1 is executed before R_2.

4 Protocol Overview

Droopy and Dripple are built upon existing SMR protocols and provide the same interfaces to upper applications. Dripple additionally requires the information about accessed partitions.

4.1 Droopy

Droopy is designed to dynamically reconfigure the set of leaders. Each set is called a *configuration*. Generally speaking, Droopy splits the space of sequence *indices* (e.g., logical sequence numbers or physical clocks) into ranges (equal to, e.g., δ sequence numbers or seconds). Each range of indices is mapped to a configuration and maintained as *a lease*. Although a lease can reflect a physical time range in clock-based systems, time anomalies such as clock drifts, do not affect the correctness of Droopy, but only its performance (as we prove in Sec. 5.3.1).

In a nutshell, a lease in Droopy is a commitment of a subset of replicas to a range of sequence indices, such that only those specific replicas, i.e., the leaders, can propose requests. Leases are proposed by replicas and maintained by a total order primitive, which can be implemented by a classical single-leader Paxos (we call this *Lease-Paxos* or simply *L-Paxos*). In a sense, Droopy follows the approach to state machine reconfiguration proposed in [23]. Droopy however targets the optimization of perceived latency and focuses on the reconfiguration of leader set.

We assume that the length of each lease is δ. When current lease (with lease number ln) is about to expire, i.e., the length of available sequence indices is less than λ (refer to Fig. 3), at least $t+1$ replicas propose a new leader configuration for the next lease $ln+1$, so that the crash of any t replicas can not stop the lease renewal process. The leader set in each lease is selected based on previous workloads and network condition. In this article we assume that the goal for Droopy is to minimize the average latency across all sites. Each lease should be consistent among all replicas. This is guaranteed by a consensus primitive within L-Paxos.

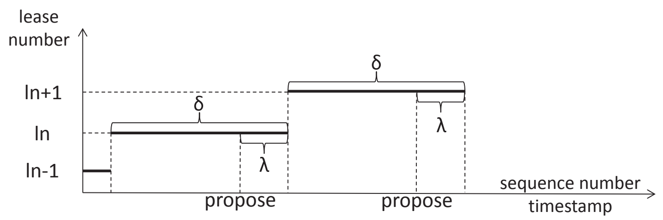


Fig. 3: Lease renewal.

A client submits its request by contacting the nearby replica, that we call the *source replica* of the request. Upon reception of a request, the source replica proposes the request if it is in the leader set in current lease. Otherwise, the source replica propagates the request to a leader that acts as a *proxy*. For each non-leader replica, its proxy is the leader that is expected to introduce the minimum commit latency, with respect to the non-leader replica. In default, upon a request is committed and executed, the source replica sends a reply to the client. Each replica also updates its *frequency array* monitor, that records the number of requests from each source replica. Frequency array serves to help the decision of next leader set. Each replica also periodically measures and shares round-trip latencies from itself to others for the same purpose.

4.2 Dripple

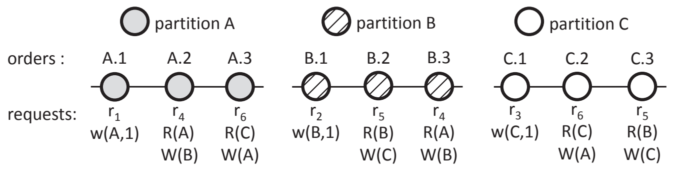
Dripple divides the system state (a.k.a., objects) into multiple partitions, and process cross-partition requests wisely so that strong consistency is still preserved. By deploying Dripple, each partition can be maintained by an individual Droopy instance in order to optimize latency per partition. Requests accessing distinct partitions can be executed in different order.

Existing leaderless protocols [7], [9], [16] detect and order non-commutative requests *dynamically* by introducing more message exchanges. In a similar way, Generic Broadcast [24] is a broadcast primitive that explores conflict relation among messages and only orders conflicting messages. Upon detecting a conflict, Generic Broadcast launches a consensus instance to resolve it. Whereas, Dripple divides the system state *in advance* so that non-commutative requests are grouped before being proposed. By doing this, non-commutative requests are ordered in the same partition by a single leader or all-leader protocols, which usually have more efficient message patterns than leaderless protocols if conflicts exist.

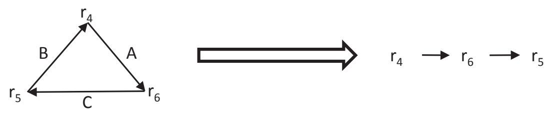
Dripple is also different from other state-partitioning schemes [25], [26], [27], which are particularly designed for providing scalability by partial replication. Instead, each partition in Dripple is replicated at every replica (in line with SMR). Hence, dependencies among different partitions can be resolved locally at each replica, without additional communications among partitions. Nonetheless, applications can easily employ existing state-partitioning approaches to improve scalability.

When Dripple is enabled, a client submits its request by contacting its source replica just like in Droopy. The source replica treats Droopy enabled SMR as a black-box and proposes the request in each partition that the request is going to access (i.e., read or write). Upon a request is committed in every involved partition, each replica tries to execute the request by dynamically constructing a *minimum execution set* (MES), in order to preserve strong consistency across partitions. Roughly speaking, MES is a *minimum* set of requests, in which every pair of requests have mutual dependence. Requests in MES are executed sequentially based on a deterministic order. Nevertheless, commutative requests should be included in distinct MES.

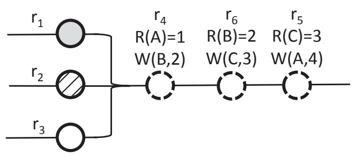
An illustrative example of MES is shown in Fig. 4. There are 3 partitions, A, B and C. The order in each partition is given by the format $partitionName.sequenceIndex$ (e.g., in partition A, $A.1 < A.2 < A.3$). The committed orders of 6 requests in these 3 partitions are presented in Fig. 4a. For example, request r_1 is committed at $A.1$, which is prior to request r_4 committed at $A.2$. Since requests r_1, r_2 and r_3 access a single and distinct partition, each of them can constitute a MES and can be executed independently. In contrast, requests r_4, r_5 and r_6 form a dependency loop as shown in Fig. 4b: r_4 is committed before r_6 in partition A; r_6 is committed before r_5 in partition C; and, r_5 is committed before r_4 in partition B. As shown in Fig. 4b, the deterministic order is $r_4 \rightarrow r_6 \rightarrow r_5$. Eventually, as shown in Fig. 4c, r_1, r_2 and r_3 are executed in parallel, then r_4, r_6 and r_5 are executed sequentially in a single MES afterwards.



(a) Logical orders committed in partition A, B and C.



(b) Minimum execution set.



(c) Execution order.

Fig. 4: Dripple example.

4.3 Fault-Tolerance

Since Droopy and Dripple are built upon existing single-leader and all-leader protocols, they can rely on underlying protocols to tolerate faults. More specifically, Paxos and Mencius use a consensus algorithm [5] to guarantee reliability. Consensus ensures that any request committed at a given sequence number will never be replaced. Clock-RSM makes use of a reconfiguration protocol to remove faulty replicas from current membership. In some sense, our approaches improve the way by which requests are ordered, not the way that makes requests reliable.

5 Protocol Details

5.1 Droopy

Object definitions given in Alg. 1 are used in Droopy pseudocode, which is given in Alg. 2.

Proposing. To issue request req, client c sends req to the closest replica s_i. Upon receiving req from c (line 1, Alg. 2), s_i becomes the source replica of req (line 2). Then, s_i obtains a sequence index sn by invocation GETORDER() provided by an interface to the underlying SMR protocol (see line 3 in Alg. 2, as well as lines 1-2 in Alg. 3). s_i may update the lease number if necessary (lines 4-5). If s_i is one of the leaders in current lease ln, i.e., $s_i \in config_{ln}$, then s_i proposes req with the sn obtained; otherwise, s_i propagates req to its proxy $p_i \in config_{ln}$, which is expected to introduce the minimum commit latency with respect to s_i (lines 6-9). Note that proxy p_i may not be the closest replica regarding s_i. In

Algorithm 1 object definitions.

c: client
s_i, s_j, s_k : replicas
req : request from client c
src : source replica
sn : sequence index
ln : current lease number
LE_{ln} : end index of lease ln
$config_{ln}$: the leader set in lease ln
$replicas$: the set of n replicas in the system
$clock$: physical clock at replica s_i
$d_{*,*}$: latency table (from all replicas to all replicas)
$freq_*$: the number of requests received by each source replica
$latest_*$: the most recent sequence indices updated from each replica

Algorithm 2 Droopy: Dynamic reconfiguration at replica s_i.

```
    abstract function GETORDER()
    abstract function GETNEWCONFIG($freq, d$)

 1: upon receive $\langle$REQUEST$, req, [s_j]\rangle$ from client $c$ or $s_k$ do
 2:     $src \leftarrow (s_j = null)\ ?\ s_i : s_j$                /* source replica */
 3:     $sn \leftarrow$ GETORDER()
 4:     while $sn \geq LE_{ln}$ do
 5:         $ln \leftarrow ln + 1$
 6:     if $s_i \in config_{ln}$ then                       /* leader */
 7:         PROPOSE($req, sn, src$)
 8:     else                                             /* non-leader */
 9:         sends $\langle$REQUEST$, req, src\rangle$ to $p_i \in config_{ln}$

10: upon DECIDE($req, sn, src$) and UPDATED($sn$) is $true$ and
    all preceding requests are executed do
11:     $rep \leftarrow$ execute $req$
12:     $freq_{src} \leftarrow freq_{src} + 1$
13:     if $s_i = src$ then      /* source replica replies to client */
14:         send $\langle$REPLY$, rep\rangle$ to client $c$
15:     if $LE_{ln} - \lambda \leq sn$      /* time to propose a new lease */
16:         $config \leftarrow$ GETNEWCONFIG($freq, d$)
17:         PROPOSE($ln + 1$,$config$,$LE_{ln} + \delta$) in L-Paxos
18:         reset $freq_*$

19: upon DECIDE($ln'$,$config$,$LE$) in L-Paxos do
20:     $config_{ln'} \leftarrow config$
21:     $LE_{ln'} \leftarrow LE$

22: function UPDATED($sn$)                      /* returns boolean */
23:     return $\forall ln$ and $\forall s_k \in config_{ln}$ : $latest[s_k] \geq LE_{ln}$ or
    $latest[s_k] \geq sn$
```

a rare case, req might be propagated several times, hence $s_j \neq s_k$ can happen in line 1.

Committing. In commit phase, Droopy needs to modify the underlying SMR protocol at the condition which checks whether there are still requests preceding sn. After confirming this, each replica can further commit and execute req if all previous requests are committed and executed. In particular, this check in all-leader protocols involves all replicas (leaders), whereas in Paxos it involves the single leader (which is trivial with FIFO assumption). In Droopy, this condition depends on the leader set in each lease. The modification is shown as function UPDATED in Alg. 2. More specifically (line 23), UPDATED (i.e., whether no request precedes req) returns true if $\forall ln$ and $\forall s_k \in config_{ln}$, either lease ln is not available anymore at leader s_k (i.e., $latest[s_k] \geq LE_{ln}$), or the most recent received index from s_k is not smaller than sn (i.e., $latest[s_k] \geq sn$).

Upon req is committed at replica s_i (line 10), i.e., req is replicated by a majority of replicas and all preceding requests are executed locally at s_i, s_i (1) executes req, and replies response to client c if s_i is the source replica, and (2) updates frequency array f (lines 11-14).

Lease update. (also refer to Fig. 3) When current lease ln is about to expire (i.e., less than λ sequence indices are available, lines 15-18 in Alg. 2), replica s_i first selects a new leader set $config$ based on frequency array f_* and latency table $d_{*,*}$; then s_i proposes $config$ piggybacked with lease number $ln+1$ in L-Paxos and resets $freq_*$. Upon $config$ in lease ln' is decided at replica s_i (lines 19-21), s_i updates the leader set $config_{ln'}$ and the end index $LE_{ln'}$.

Latency update. (ignored in Alg. 2) Replica s_i periodically measures the round-trip latency from itself to others (e.g., by using ping utility). Then, s_i updates the latency table $d_{i,*}$ and shares $d_{i,*}$ with other replicas by broadcasting message $\langle \text{LATENCY}, i, d_{i,*}\rangle$. Upon receiving $\langle \text{LATENCY}, i, d\rangle$, replica s_j replaces local variable $d_{i,*}$ by d.

Configuration selection. To select an appropriate configuration for the next lease, replica s_i enumerates all possible combinations of leader sets by function GETNEWCONFIG (line 16).[3] Given a leader set, s_i estimates an average latency based on latency table $d_{*,*}$ and frequency array f_*. For example, in Amazon globally distributed platform (as shown in Fig. 5), if the workload is located at US East only, then GETNEWCONFIG should return US East as the single leader.

The calculation of estimated latency depends on the internals of an underlying SMR protocol proceeds. For instance, to confirm that all preceding requests have been received, one-way latency from the farthest replica matters in Clock-RSM; whereas in Mencius, this latency can be (at most) a round-trip.

More specifically, the latency for request req is dominated by three conditions: ① the time it takes for req to be proposed by a leader and further replicated by a majority; ② the time it takes for source replica s_i to confirm that no request with a sequence index smaller than req will be proposed by any leader (especially by the farthest leader); and ③ the time it takes for source replica s_i to commit all requests preceding req.

Algorithm 3 Clock-RSM function.

```
1: function GETORDER()
2:     return $clock$                                /* physical clock */

3: function GETNEWCONFIG($freq, d$)
4:     return $s \subseteq replicas$ s.t.
       $min(\sum_{i=1}^{n} freq_i \times min( max\left\{\begin{array}{c} \textcircled{1} d_{i,j} + median(d_{j,k} + d_{k,i} | \forall s_k) \\ \textcircled{2} d_{i,j} + max(d_{k,i} | \forall s_k \in s) \\ \textcircled{3} d_{i,j} + max((median(d_{k,l} + d_{l,i}) | \forall s_l) | \forall s_k \in s) \end{array}\right\} | \forall s_j \in s) | \forall s \subseteq replicas)$
```

Based on these three conditions, calculation for each specific SMR protocol can be (and should be) designed individually. An illustrative calculation for Clock-RSM is shown in line 4 in Alg. 3. Set s enumerates the number of possible leader sets. Index i and j enumerate the number of source replicas and the number of leaders, respectively. For each given source replica and leader, term ① calculates the (estimated) latency to send req from s_i to its proxy s_j (i.e., $d_{i,j}$) plus the latency to replicate req by a majority and inform s_i finally. Term ② calculates the latency to notify s_i (from $\forall s_k \in s$) that no leader will propose a preceding request, upon s_j proposes req (hence $d_{i,j}$ is added at first). Term ③ considers the worst case that every other leader concurrently proposes a preceding request when s_j is proposing req. Note that this term can be further refined (in future work), e.g., by considering the interval of consecutive proposals from each leader.

3. For typical small values of n, a brute force algorithm is acceptable. Exceptionally with a large n, one can make use of a greedy algorithm instead.

5.2 Dripple

To simplify the exposition, we assume that each Dripple partition contains exactly one object. Nevertheless, in practice highly correlated objects should be grouped into one partition. Either application developer knows in advance the approximate correlation among objects, thus the total number of partitions as well as the mapping from each object to its partition can be pre-defined; or, partitions can be dynamically adjusted, e.g., by supporting MERGE and DIVIDE operations — the latter mechanism is out of the scope and is an interesting direction for the avenue of future work.

We say request req is proposed or committed in partition p if req is proposed or committed by the replicated state machine that manages partition p.

Proposing. Client c issues request req by sending req to its source replica s_i (just like in Droopy). Upon receiving req (in Alg. 4 lines 1-2), s_i determines which partitions of state req will access; these comprise the *access set* of req, denoted by $set(req)$. This step is accomplished by interacting with upper applications, e.g., through an upper-call provided by applications. For example, a transactional system by this interaction returns the set of partitions that contains the keys that req will read or write. In order to guarantee strict serializability [13] among all requests, $set(req)$ should not exclude any object that will be actually accessed, but can be a superset of them.

Then, s_i processes req individually in each partition in $set(req)$ as described in Droopy (in Alg. 4 lines 3-4). Note that this step can be processed in parallel for different partitions.

Committing. Upon req is committed in partition obj at replica s_i, s_i inserts req into a *committed queue* for obj, i.e., $CmtReq_{obj}$ (lines 5-6).

Upon req is committed in every partition in $set(req)$ (lines 7-8), i.e., $\forall obj \in set(req) : req \in CmtReq_{obj}$, s_i inserts req into an ordered set $OrderedReq$. We say req is *ordered* in this case. Note that req can be executed only if req is ordered.

Executing. In order to guarantee strict serializability, execution of ordered requests should follow logical orders defined by sequence indices in every accessed partition. Without cross-partition coordination, execution of a request that accesses multiple partitions may violate serializability. For instance, if $req = \{write(A,1), write(B,2)\}$

Algorithm 4 Dripple: multi-partition commit at replica s_i.

```
M: minimum execution set
set(req): the set of objects that req accesses
CmtReq_obj: the queue of committed requests that access obj
OrderedReq: {req|∀obj : obj ∈ set(req) → req ∈ CmtReq_obj}
ExReq: the set of requests that have been executed

1:  upon receive ⟨REQUEST, req⟩ from client c do
2:      obtain set(req)
3:      for ∀ obj ∈ set(req) do
4:          PROPOSE(req, s_i) in partition obj (in parallel)

5:  upon COMMIT(req, s_j) in partition obj do
6:      append req to CmtReq_obj
7:      if ORDERED(req) then
8:          append req to OrderedReq

9:  upon ∃ a set of requests M ⊆ OrderedReq s.t.
    (1) ∀req ∈ M s.t. if ∃req': PRE(req',req), then either req' ∈
    M or req' ∈ ExReq
    (2) ∄M' ⊂ M: M' satisfies (1) do
10:     execute ∀req ∈ M based on a deterministic order
11:     remove ∀req ∈ M from OrderedReq and CmtReq_*
12:     add req to ExReq

13: function ORDERED(req)
14:     return ∀obj ∈ set(req) : req ∈ CmtReq_obj

15: function PRE(req,req')
16:     return ∃obj : req precedes req' in CmtReq_obj
```

(i.e., change object A to 1 and B to 2) precedes $req' = \{write(A,2), write(B,1)\}$ in partition A and req' precedes req in partition B, then two replicas may execute req and req' in opposite direction. We further define a relationship between two requests in the following.

Definition 2. *Request req' **directly precedes** req, or* PRE*(req',req), if $\exists obj$ s.t. req' directly precedes req in partition obj, i.e., $\exists k$ s.t. $CmtReq_{obj}[k] = req'$ and $CmtReq_{obj}[k{+}1] = req$.*

We also say req' is a directly preceding request of req. To execute ordered requests (line 9), replica s_i selects a set $s \subseteq OrderedReq$, which is named *minimum execution set* and satisfies the following conditions.

Definition 3. *A **minimum execution set** $\mathcal{M}$ is a set of ordered requests which satisfies*

(1) $\forall req \in \mathcal{M}$ s.t. if $\exists req'$: PRE*(req',req), then either $req' \in \mathcal{M}$ or $req' \in ExReq$;*

(2) $\nexists \mathcal{M}' \subset \mathcal{M}$: $\mathcal{M}'$ satisfies (1).

The first condition guarantees that each of req's directly preceding requests is either in $\mathcal{M}$ or in $ExReq$ (i.e., executed already). The second condition ensures that every replica selects the same $\mathcal{M}$ for each given req.

Then, replica s_i executes requests in $\mathcal{M}$ sequentially based on a deterministic order (line 10). For instance, a simple scheme is executing requests in $\mathcal{M}$ based on their unique identifiers ($timestamp.clientId$).

Finally, s_i removes all executed requests from $OrderedReq$ and involved $CmtReq_*$ (line 11), and adds executed requests to the set $ExReq$ (line 12).

Read-only requests. As discussed in some existing SMR protocols, read-only requests can be optimized particularly by read leases [28], [29]. With Dripple, a read-only request that involves only one partition can be executed locally by any replica which holds the corresponding read lease. However, if a read-only request involves more than one partition, the request has to be treated and executed as a normal read-write request.

5.3 Proof

In this section we prove the correctness of Droopy and Dripple. More specifically, in Sec. 5.3.1 we prove that Droopy preserves linearizability, i.e., a total order among all requests. Then, in Sec. 5.3.2 we prove that Dripple guarantees serializability across partitions. Real-time order in strict serializability is trivially ensured by durability property of replicated state machine.

5.3.1 Linearizability

Since we build Droopy on top of existing SMR protocols, without leader reconfiguration we simply run the original single-leader or all-leader protocol, i.e., a single lease is in operation endlessly, in which a single replica or every replica acts as a leader. The original protocols (e.g., Paxos, Mencius or Clock-RSM) have the following theorem.

Theorem 1. *If replica s_i has executed a request at sequence index sn, then s_i must have executed all requests with sequence indices smaller than sn.*

In other words, requests are executed in order based on their sequence indices. We argue that Theorem 1 is still true when enabling Droopy.

As we described in Sec. 4.1 and 5.1, LE_{ln} indicates the end index of lease ln. LE_{ln} is determined by L-Paxos, so based on reliability of classical Paxos, every replica should receive the same LE_{ln} (eventually). Before acknowledging LE_{ln+1} and the leader set in lease $ln+1$, replica s_i will not execute any request with sequence index larger than LE_{ln}.

In a proof by contradiction we assume that replica s_i has executed request req at sequence index sn ($LE_{ln-1} \leq sn < LE_{ln}$), but has not executed request req' at sequence index $sn' < sn$. Either $sn' < LE_{ln-1}$, then based on line 24 in Alg. 2[4], s_i must have confirmed that every leader in lease $ln' < ln$ has passed $LE_{ln'}$ and s_i has received every proposal (by FIFO assumption), i.e., $\forall ln' < ln$ and $\forall s_k \in config_{ln'}$: $lastest[s_k] \geq LE_{ln'}$; or $sn' \geq LE_{ln-1}$, then based on line 24, s_i must have confirmed that every leader in ln has passed sn, i.e., $\forall s_k \in config_{ln} : sn \leq lastest[s_k]$. In either case, no req' exists or s_i must have executed req'.

5.3.2 Serializability

Then we prove that Dripple ensures serializability. More specifically, we prove if eventually two replicas s_i and s_i' have executed all requests[5] and no request is further issued, then the execution order at these two replicas is equivalent to the same sequential order.

At first, we construct a directed graph $G = (V, E)$ based on a minimum execution set $\mathcal{M}$, where $V = \mathcal{M}$ and E

4. Line 24 in Alg. 2 is written in a combined mode.
5. This assumption is guaranteed by availability property of replicated state machine deployed in each partition.

contains all PRE relations between requests in V. Namely, if PRE(req,req') and $req, req' \in \mathcal{M}$, then there is an edge from req to req' in E. Then we prove:

Lemma 1. *$G = (V, E)$ is strongly connected.*

A directed graph $G = (V, E)$ is strongly connected if $\forall u, v \in V$, there is a path or "walk" from u to v, which is composed by edges in E.

We prove it by contradiction. $\forall req, req' \in V$, if there is no path from req to req', then we can assume there exists a subset $V_1 \subset V$ that req can walk to (by edges in E) and there exists another subset $V_2 \subset V$ that req can not walk to. Obviously, $req \in V_1$ and $req' \in V_2$. Furthermore, there is no PRE relation from any request req_1 in V_1 to any request req_2 in V_2, otherwise req_2 should be included in V_1. Hence, it's easy to see that V_2 satisfies Def. 3 (1) as well. Therefore, Def. 3 (2) is violated.

Then we prove

Lemma 2. *if replica s_i has executed a collection C of k minimum execution sets and replica s'_i has executed a collection C' of k' minimum execution sets, then $C = C'$.*

Assume $\exists s \in C$ and $\exists req, req' \in \mathcal{M}$, i.e., by replica s_i req and req' are executed within the same minimum execution set $\mathcal{M}$. Based on Lemma 1, there exists a dependency path from req to req' and vice versa. Obviously, $\exists \mathcal{M}' \in C'$ satisfying $req \in \mathcal{M}'$. Then based on Definition 3 (1), req' must be in $\mathcal{M}'$ (it's impossible that req' is executed prior to $\mathcal{M}'$ or after $\mathcal{M}'$ since there is a dependency path from req' to req and vice versa, and $req \in \mathcal{M}'$).

Although $C = C'$, different replicas may execute minimum execution sets out of order. Now we prove by induction that

Lemma 3. *the execution order at each replica incrementally forms a Directed Acyclic Graph (DAG).*

A DAG [30] is a directed graph in which there is no directed cycle, i.e., there is no path that starts from any node $v \in V$ and goes through several edges in E and comes back to v. By the property of linear extension of DAG, different execution orders that follow the same DAG are equivalent, even if independent requests are executed out of order.

We consider the state after executing i minimum execution sets (no order is required here) as a directed graph $G(V_i, E_i)$ ($i \geq 0$), in which the set of nodes V_i contains all executed requests and the set of edges E_i contains all PRE relations among requests in V_i.

Each replica starts from a initial state with no request executed. We denote the initial state as $G(V_0, E_0)$, i.e., $V_0 = \emptyset$ and $E_0 = \emptyset$. Obviously, $G(V_0, E_0)$ is a DAG.

We assume $G(V_i, E_i)$ ($i \geq 0$) is a DAG.

Upon the $(i+1)th$ minimum execution set $g = G(V, E)$ is ready to execute, either $|V| = 1$ (i.e., there is only one request in V), hence all of its directly preceding requests have been executed already (i.e., in V_i); or $|V| > 1$, then g is linearized deterministically into $G(V, E')$. By combining V_i and V, and by combining E_i, E', and the set of all PRE relations from requests in V_i to requests in V, we form a new graph $G(V_{i+1}, E_{i+1})$. It's easy to observe that $G(V_{i+1}, E_{i+1})$ is still a DAG, since we combine one DAG, i.e., $G(V_i, E_i)$, with another DAG, i.e., $G(V, E')$, plus the added edges are only in one direction, i.e., from V_i to V.

By Lemma 2, replicas s_i and s'_i have executed the same collection of minimum execution sets. PRE relations within a minimum execution set are decided deterministically based on request identifiers. Whereas, PRE relations among different minimum execution sets are determined by replicated state machine in each partition. Then by Lemma 3 and by property of DAG, the execution orders at s_i and s'_i are equivalent to the same sequential order.

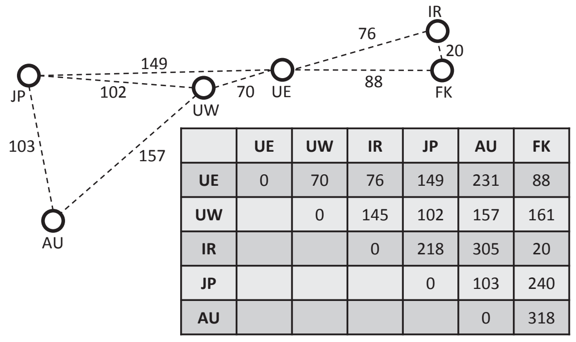
6 Experimental evaluation

In this section we evaluate the performance of D^{2}Clock-RSM and compare it to state-of-the-art SMR protocols using the Amazon EC2 worldwide cloud platform. We start by presenting the experimental setup in Sec. 6.1. We then test the perceived latency of Droopy enabled Clock-RSM (simply Droopy) and compare it to native Clock-RSM and Paxos under imbalanced workloads (Sec. 6.2). Finally, by enabling Dripple as well, we compare D^{2}Clock-RSM with state-of-the-art protocols under three different types of balanced workloads (in Sec. 6.3).

6.1 Experiment setup

Both replicas and clients are deployed in instances on Amazon EC2 platform that comprises widely distributed datacenters, interconnected by the Internet. In each datacenter there is one virtual machine dedicated to one replica and one virtual machine for all clients.

We run the experiments on mid-range virtual machines named "c4.large" instances that contain 2 vCPUs, 3.75 GB of memory. The round-trip latencies among 6 EC2 sites (measured by ping utility) is shown in Fig. 5.



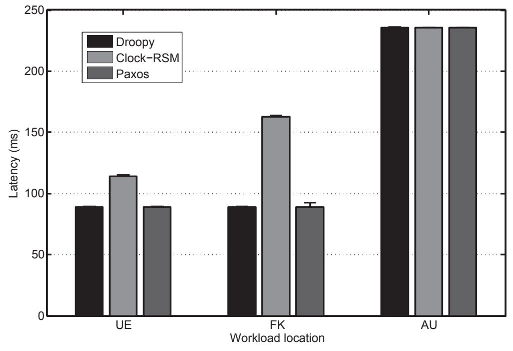
	UE	UW	IR	JP	AU	FK
UE	0	70	76	149	231	88
UW		0	145	102	157	161
IR			0	218	305	20
JP				0	103	240
AU					0	318

Fig. 5: Round-trip latencies (ms) among 6 Amazon EC2 sites used in our experiments: US East (UE), US West (UW), Ireland (IR), Japan (JP), Australia (AU) and Frankfurt (FK).

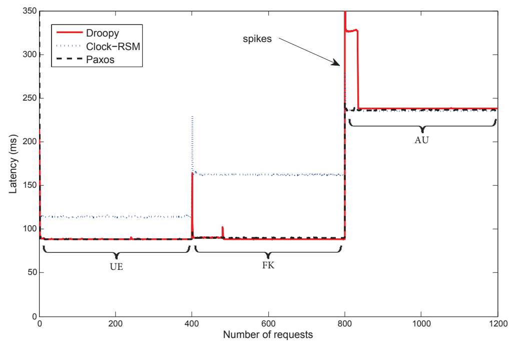
In Droopy, we set δ to 10 seconds and λ to 2 seconds, i.e., the length of each lease is 10 seconds, and 2 seconds before expiration, each replica proposes a new configuration for the next lease. We believe these two settings are sufficiently small to demonstrate the usability of Droopy.

6.2 Imbalanced workloads

To emphasize the benefits we can obtain by enabling Droopy, we first run Droopy enabled Clock-RSM under

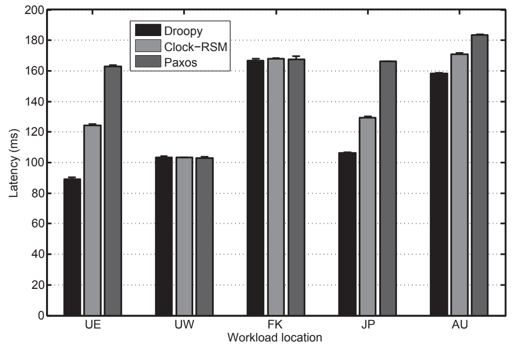


(a) Average latency (bars) and 95%ile latency (lines atop bars).

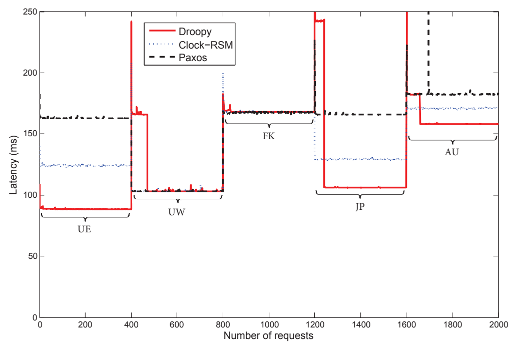


(b) Latency change when the workload is moved from site to site; the leader set is reconfigured as follows: {UE}→{UE,FK}→{FK}→{FK,AU}→{AU}.

Fig. 6: Latency (in milliseconds, lower is better) of Droopy, Clock-RSM and Paxos under imbalanced workloads when $t = 1$.



(a) Average latency (bars) and 95%ile latency (lines atop bars).



(b) Latency change when the workload is moved from site to site; the leader set is reconfigured as follows: {UE}→{UE,UW}→{UW}→{UW,FK}→{FK}→{JP}→{JP,AU}→ {AU}.

Fig. 7: Latency of Droopy, Clock-RSM and Paxos under imbalanced workloads when $t = 2$.

imbalanced workloads, i.e., only clients at one site are issuing requests at a time. We then compare Droopy to the native Clock-RSM and Paxos. In this experiment, 40 clients at each specific site issue requests of 64B to their source replica in closed-loop.

We deploy the experiments at US East (UE), Frankfurt (FK) and Australia (AU) for $t = 1$ case, and UE, US West (UW), FK, Japan (JP) and AU for $t = 2$ case. For a fair comparison, we position the single leader of Paxos in the (geographical) middle of all replicas. Thus, when $t = 1$ the single leader is located at UE; when $t = 2$ the single leader is located at UW.

The results for $t = 1$ and $t = 2$ are shown in Fig. 6 and Fig. 7, respectively. Notice that, to clearly demonstrate Droopy's latency in a steady state, in Fig. 6a and Fig. 7a we manually configure the leader set to the same site as the workload (i.e., clients) locates, and we run the same experiment site by site. Therefore, in this setup there is no need to reconfigure the leader set. In Fig. 6b and Fig. 7b we automatically change the location of active clients/workload from site to site and show how the leader set is reconfigured and its influence on latency. More specifically, upon 40 clients located at one site have executed all requests, clients at another site start issuing requests.

$t = 1$. In Fig. 6a we show the average latency and the 95%ile latency at each site for all protocols. When the workload is located at UE and FK, we observe that Droopy effectively reduces latency compared to the native Clock-RSM. However, there is no improvement at AU, since replica at AU is positioned very far from another two replicas and AU actually introduces delayed commit problem for UE and FK. By comparing Fig. 6a with Fig. 5 we also observe that both Paxos and Droopy achieve the optimal latency (i.e., a round-trip delay from the source replica to a majority). The set of UE, FK and AU can be considered as a special configuration since UE (where the single leader of Paxos locates) is placed between FK and AU, hence in any case it is (more) efficient to replicate requests at UE.

Fig. 6b demonstrates more clearly how latency is affected by workload movement and leader reconfiguration. Interestingly, as shown in Fig. 6b, a latency spike occurs when the workload is moved from FK to AU. This is because the leader set has not yet been adjusted towards the new workload, i.e., first the leader is placed at FK, then reconfigured to {FK,AU} as an intermediate state, and finally reconfigured to AU. No latency spike happened when we moved the workload from UE to FK. This is because sending requests from FK (where the new workload locates) to UE (where the old leader locates) and making them replicated by a majority (i.e., by UE and FK) is as efficient as a round-trip communication from FK to UE (in case FK is the leader).

$t = 2$. In Fig. 7a, when the workload is deployed at UE, JP and AU, we observe that Droopy's latency is reduced compared to the native Clock-RSM. In Clock-RSM, the latency achieved at JP and AU is higher because of the delayed commit problem introduced by FK. And, similarly to the $t = 1$ case, replica AU causes the delayed commit problem for the workload at UE.

Paxos has the higher latency compared to the other two protocols, except when the workloads are deployed at UW and FK. The reasons are different. UW achieves low latency simply because the single leader (in Paxos) locates at the same site. Whereas, for clients at FK, propagating requests to the leader (UW) and making them replicated by a majority (by UW, UE and FK in this case) is as efficient as making requests proposed by FK (in Droopy and Clock-RSM) and replicated by a majority (by UW, UE and FK), since UE is positioned between UW and FK (as shown in Fig. 5).

Fig. 6a and Fig. 7a also verify our expectations that (1) *neither* single-leader *nor* all-leader SMR can achieve the minimum latency in all cases; and, (2) statically fixing the leader set at one or more replicas cannot achieve the minimum latency in all cases. For example, replica AU should be excluded from the leader set when the workload locates only at UE. However, AU should be included in the leader set when the workload locates only at AU. These two observations practically motivate Droopy, our leader reconfiguration protocol.

In Fig. 7b, we also observe that a latency spike occurs whenever the workload is moved from one site to another (except for the movement from UW to FK, as for the same reason explained in $t = 1$ case), due to the delay of leader reconfiguration.

Partially imbalanced workloads. To more comprehensively evaluate Droopy, we further deploy the three protocols under partially imbalanced workloads, where 40 clients at UE and 10 clients at AU issuing 64B requests to their source replicas in closed-loop. In Fig. 8 we show the latency distribution at two replicas (UE and AU) for each protocol. Interestingly, under these workloads, Droopy reconfigures the leader set into {UE,JP}, not exactly the sites where the workloads are deployed (i.e., UE and AU). At site UE, Droopy achieves much lower latency than Clock-RSM, whereas at site AU Droopy has higher latency than Clock-RSM.

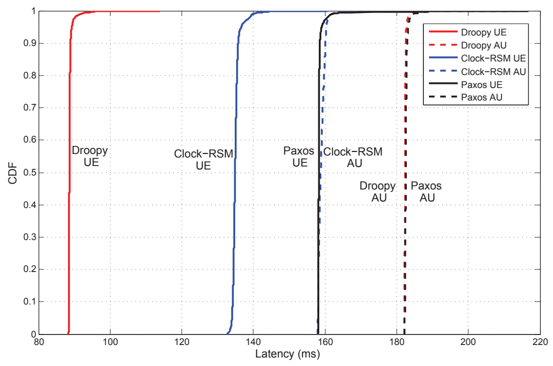


Fig. 8: Latency distribution under partially imbalanced workloads ($t = 2$), where 40 clients at UE and 10 clients at AU issue requests. The single leader of Paxos locates at UW. The leader set is reconfigured to {UE,JP} in Droopy.

This choice is made reasonably. If AU is one of the leaders, then extra delay introduced by AU can increase the latency at UE dramatically. Since 4 times of clients are deployed at UE compared to that of AU, selecting JP as a leader (instead of AU) can at the same time mitigate the delayed commit problem and keep one of leaders close to AU. Besides, we also observe from Fig. 8 that at AU Droopy achieves similar latency compared to Paxos. This is because requests from AU should be propagated to JP at first (as the proxy of AU), then proposed by JP and replicated by UW, and finally committed by AU. This commitment path is similar in Paxos, albeit in the opposite direction: requests from AU are at first propagated to UW (where the single leader is placed), then proposed by UW and replicated by JP, and finally committed by AU.

We further deploy the three protocols under another partially imbalanced workloads, where 40 clients at UE and 5 clients at each other site issue requests to their source replicas. The average and the 95%ile latency is shown in Fig. 9. Although the workloads are deployed at every site in this case, Droopy made the same choice: configuring its leader set to {UE,JP}. By comparing the three protocols at each site in Fig. 9 we can more clearly observe the "priority" behind them: (1) Paxos optimizes the latency at a given site (UW in this case); (2) Clock-RSM tries to "balance" the latency at every site regardless of the workloads; and, (3) Droopy tries to achieve the optimal average latency based on previous workloads.

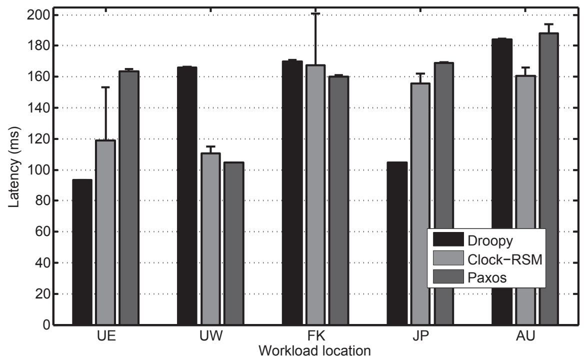
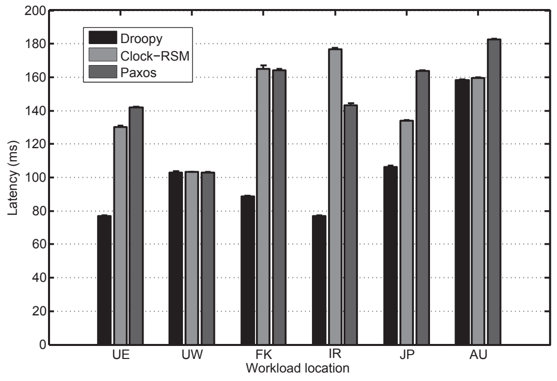


Fig. 9: Average latency (bars) and 95%ile latency (lines atop bars) when 40 clients at UE and 5 clients at each other site issue requests. The single leader in Paxos is located at UW. The leader set is configured to {UE,JP} in Droopy.

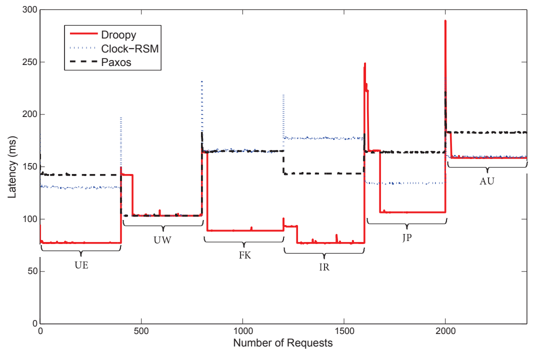
Spare replica. As discussed in [31], adding spare replicas to a geo-replicated state machine can sometimes (depending on replica location and workload) dramatically reduce latency, compared to the standard configuration where $n = 2t + 1$. Obviously, this can also give Droopy more choices in selecting a leader set. We add one more replica (Ireland, IR) to $t = 2$ configuration, i.e., $n = 6$ at present, and evaluate how this modification affects the performance of Droopy, Clock-RSM and Paxos.

We deploy the three protocols under imbalanced workloads again. The average latency and the latency change with the workload movement is shown in Fig. 10. At site UE, FK, and IR we observe a more significant improvement achieved by Droopy, compared to Fig. 7. This is because introducing spare replica IR makes nearby replicas UE and FK more efficiently to replicate their requests by $t + 1$

replicas[6], which indirectly enlarges delayed commit caused by distant replicas (AU and JP). Compared to Fig. 7a, the latency of Paxos at UE is reduced as well because of the same reason: requests proposed by UE can be replicated by UW and IR more easily than by UW and FK (when $n = 2t + 1$).



(a) Average latency (bars) and 95%ile latency (lines atop bars).



(b) Latency change when the workload is moved from site to site; the leader set is reconfigured as follows: {UE}→{UE,UW}→{UW}→{UW,FK}→{FK}→{FK,IR}→{IR} →{IR,JP} →{JP}→{JP,AU}→ {AU}.

Fig. 10: Latency of Droopy, Clock-RSM and Paxos under imbalanced workloads where $t = 2$ and $n = 6$.

Besides, in Clock-RSM the average latency at IR is higher than we expect, based on the measurement in Fig. 5. The round-trip latency from AU to IR (measured by ping utility) is 305 ms, hence coordination cost in Clock-RSM, i.e., one-way latency from AU to IR should be around 152 ms, rather than 176 ms measured at site IR in Fig. 10a. This unexpected higher latency is either because (1) physical clock at IR is not tightly synchronized with those of other sites; or because (2) one-way latency from some site (most possibly AU) to IR is not equal to half of its round-trip value. We can not trivially distinguish these two reasons since we can not confirm if the physical clocks at these replicas are tightly synchronized or not. However, this exceptional cost further amplifies the importance of Droopy, i.e., removing idle (and distant) replicas from the leader set, so that the long-time coordination introduced by clock asynchrony can be avoided.

6. rather than by a majority in $n = 2t + 1$ configuration; however, to ensure intersacting quorums in Clock-RSM, either $n+1-(t+1) = t+2$ replicas should be involved in reconfiguration, which retrieves a most recent state from previous epoches; or, a weighted replication can be introduced as discussed in [31].

6.3 Balanced workloads

Finally, we evaluate Droopy and Dripple enabled Clock-RSM (i.e., D^{2}Clock-RSM) under three types of *balanced* workloads, and compare D^{2}Clock-RSM with state-of-the-art SMR protocols (Paxos, native Clock-RSM, Mencius and EPaxos). In each experiment, 40 clients at each site issue requests simultaneously to their source replicas in closed-loop.

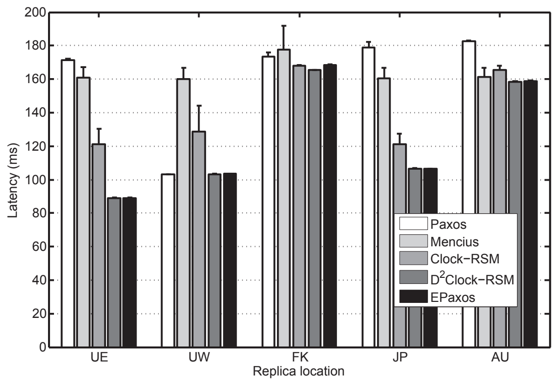
To explore request commutativity, we further prescribe that each client issues 10% of multi-partition requests and 90% of one-partition requests, where the payload for each partition is 64B. For each multi-partition request, the number of accessed partitions obeys uniform distribution, ranging from 2 to 5. We believe these settings are representative since from the perspective of applications, either most requests access a very small portion of the whole state (e.g., large-scale transactional storage system [2]); or the whole state is not suitable for partitioning. In the second case, disabling Dripple is rather efficient.

We illustrate the $t = 2$ results in Fig 11. The single leader in Paxos located at UW for a fair comparison. Note that we did not enable the optimization in EPaxos that allows replicas to respond to clients tentatively, upon a request is replicated by a fast-quorum (but not yet executed). This optimization can be applied only if the client does not expect any application-specific reply. For example, PUT operations in key-value store can be applied with this optimization, but COMMIT operations in transactional storage system can not.

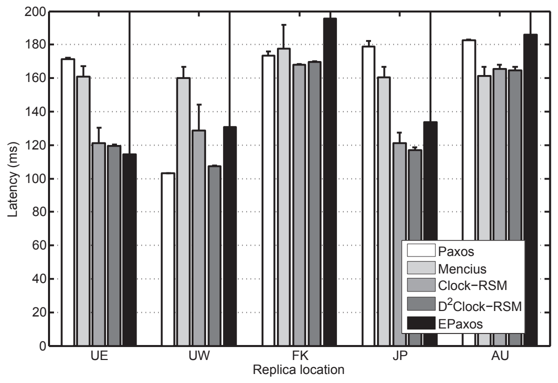
Localized workloads. We first run all protocols under localized workloads, where clients located at one site access partitions distinct from other sites. We set the total number of partitions to 1000 in D^{2}Clock-RSM (in EPaxos we set 1000 objects). For example, clients at site UE access partitions 1 to 200 randomly, clients at UW access partitions 201 to 400 randomly, and so on. We believe 1000 partitions is sufficient to demonstrate the usability of Dripple — to further provide scalability, applications could further make use of existing state-partition schemes [26].

The results under localized workloads are shown in Fig. 11a. The leader set for each partition is reconfigured to the source replica. D^{2}Clock-RSM and EPaxos have very similar performance and outperform other protocols at all sites. Localized workload can be considered as the best case for both D^{2}Clock-RSM and EPaxos. D^{2}Clock-RSM can individually reconfigure the leader set in each partition to its source replica. Whereas, in EPaxos, requests issued by different sites have no conflict among each other, hence every request can be executed as soon as the request is replicated by a fast-quorum (when $t \leq 2$ in EPaxos, a fast-quorum is equivalent to a majority). At UE, JP and AU D^{2}Clock-RSM achieves lower latency than Clock-RSM as expected, based on the Droopy experiment we tested in Sec. 6.2.

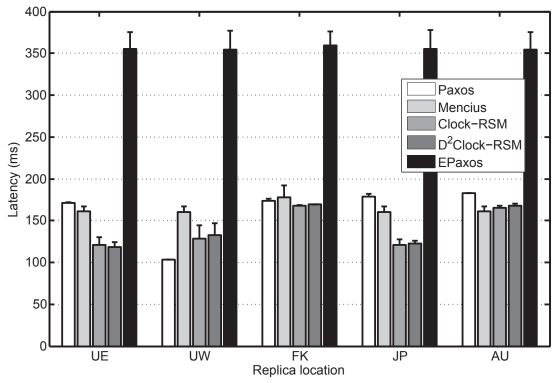
More interestingly, at site UW D^{2}Clock-RSM reduces latency as well compared to the native Clock-RSM. This is the improvement achieved by Dripple: dividing state into partitions can remove unnecessary dependencies across the requests that access distinct partitions. For example, a request



(a) Localized workloads.



(b) Uniform workloads.



(c) "Hot spot" workloads.

Fig. 11: Average latency (bars) and 95%ile latency (lines atop bars) of D^{2}Clock-RSM and state-of-the-art protocols under balanced workloads.

req proposed by UW writes partition A, and simultaneously another request req' proposed by UE writes partition B, then these two requests are committed independently in D^{2}Clock-RSM as they access distinct partitions. However, in native Clock-RSM, if req' is committed before req, UW has to wait the commitment of req' before executing req.

In Paxos the latency is dominated by the location of the single leader, i.e., UW achieves the optimal latency, whereas other sites have to suffer the additional message exchanges that propagate requests to the remote leader.

Uniform workloads. Then we deploy all protocols under uniform workloads, where each request randomly accesses one (90% probability) or 2-5 partitions (10% probability). The number of partitions in this workload is still 1000 (1000 objects for EPaxos).

The results are shown in Fig. 11b. D^{2}Clock-RSM reconfigures the leader set to all-leader configuration. In other words, each partition is simply managed by a native Clock-RSM. Under uniform workloads, the native Clock-RSM and D^{2}Clock-RSM achieve the very similar performance except for the workload deployed at UW. The reason is very similar to the one under localized workloads: Dripple removes unnecessary dependencies across distinct partitions. Although under uniform workloads partitions are randomly accessed, among 1000 partitions it is very less likely that requests proposed by UW have to wait some concurrent requests proposed by other sites (such as UE). EPaxos in this case has much higher 95%ile latency than other protocols since non-commutative requests rely on another round-trip message exchange among a majority to resolve conflicts.

"Hot spot" workloads. Finally, we deploy all protocols under the workloads which contain only 5 partitions (5 objects for EPaxos), i.e., the scenario that some small portion of state is accessed frequently by most requests. D^{2}Clock-RSM reconfigures the leader set into all-leader configuration. In this case, EPaxos has its average latency increased dramatically due to (1) the conflict resolution which relies on one more round-trip message exchange, and (2) the dense dependencies among concurrent requests (i.e., each request has to wait for a lot of other requests before being executed). These results are the worst case scenario for EPaxos as discussed in the original paper [9]. Clock-RSM and D^{2}Clock-RSM under "hot spot" workloads achieve very similar performance and outperform other protocols except for Paxos at UW, where Paxos locates its single leader at the same site and achieves the optimal latency.

7 Conclusion

We designed and implemented Droopy and Dripple, two sister approaches tailored for low-latency geo-replicated state machine. Droopy dynamically reconfigures the set of leaders based on previous workload and network condition. Dripple in turn divides the state into partitions and coordinates them wisely, so that the leader set of each partition can be reconfigured (by Droopy) individually.

By experimental evaluation on globally deployed Amazon EC2 sites we show that, Droopy efficiently reduces the perceived latency for widely distributed clients under imbalanced workloads and latencies among replicas are non-uniform. Under balanced but localized workloads, Droopy and Dripple enabled protocol outperforms existing all-leader protocols and achieves the similar performance as that of a leaderless protocol. Whereas, under uniform but request-non-commutative workloads, our approaches do not affect the performance of their native protocol and both outperform a leaderless protocol.

Acknowledgements

This work is supported in part by the EU H2020 project SUPERCLOUD (grant No. 643964) and Swiss Secretariat for Education, Research and Innovation (contract No. 15.0025). Shengyun Liu's work is supported in part by the National Key Research and Development Program (2016YFB1000101) and China Scholarship Council.

References

[1] S. Liu and M. Vukolić, "How many planet-wide leaders should there be?" *SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 3, pp. 3–6, Nov. 2015.

[2] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman *et al.*, "Spanner: Google's globally-distributed database," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 251–264.

[3] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "Mdcc: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 113–126.

[4] 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*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[5] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, pp. 133–169, May 1998.

[6] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[7] L. Lamport, "Generalized consensus and paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, March 2005.

[8] P. Sutra and M. Shapiro, "Fast genuine generalized consensus," in *2011 IEEE 30th International Symposium on Reliable Distributed Systems (SRDS)*, Oct 2011, pp. 255–264.

[9] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 358–372.

[10] 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*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 369–384.

[11] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone, "Clock-rsm: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks," in *The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, no. EPFL-CONF-198282, 2014.

[12] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[14] M. Vukolic, *Quorum Systems:With Applications to Storage and Consensus*. Morgan & Claypool, 2012.

[15] M. S. Ardekani and D. B. Terry, "A self-configurable geo-replicated cloud storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 367–381.

[16] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, pp. 79–103, 2006, 10.1007/s00446-006-0005-x.

[17] G. Santos Veronese, M. Correia, A. Bessani, and L. C. Lung, "Spin one's wheels? byzantine fault tolerance with a spinning primary," in *Reliable Distributed Systems, 2009. SRDS '09. 28th IEEE International Symposium on*, Sept 2009, pp. 135–144.

[18] G. Santos Veronese, M. Correia, A. Bessani, and L. C. Lung, "Ebawa: Efficient byzantine agreement for wide-area networks," in *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, Nov 2010, pp. 10–19.

[19] W. Wei, H. T. Gao, F. Xu, and Q. Li, "Fast mencius: Mencius with low commit latency," in *INFOCOM, 2013 Proceedings IEEE*, April 2013, pp. 881–889.

[20] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[21] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[22] "Network time protocol," http://www.ntp.org/.

[23] L. Z. Leslie Lamport, Dahlia Malkhi, "Vertical paxos and primary-backup replication," Tech. Rep., May 2009.

[24] F. Pedone and A. Schiper, "Generic broadcast," in *Proceedings of the 13th International Symposium on Distributed Computing*. London, UK, UK: Springer-Verlag, 1999, pp. 94–108.

[25] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, June 2011, pp. 454–465.

[26] C. Bezerra, F. Pedone, and R. van Renesse, "Scalable state-machine replication," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2014, pp. 331–342.

[27] R. Halalai, P. Sutra, E. Riviere, and P. Felber, "Zoofence: Principled service partitioning and application to the zookeeper coordination service," in *2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS)*, Oct 2014, pp. 67–78.

[28] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *Proceedings of the 26th annual ACM symposium on Principles of distributed computing*, ser. PODC '07. New York, NY, USA: ACM, 2007, pp. 398–407.

[29] 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*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 22:1–22:13.

[30] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[31] J. Sousa and A. Bessani, "Separating the wheat from the chaff: An empirical design for geo-replicated state machines," in *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, Sept 2015, pp. 146–155.

Dr. Shengyun Liu conducted his PhD research at EURECOM and obtained a PhD degree in Computer Science from Telecom ParisTech in 2015. He also received a MSc degree in Computer Science from National University of Defense Technology (NUDT) in 2010. He is currently a researcher at National University of Defense Technology. His research interests include Byzantine-fault tolerance, state machine replication and blockchain.

Dr. Marko Vukolić is a Research Staff Member at IBM Research - Zurich. Previously, he was a faculty at EURECOM and a visiting faculty at ETH Zurich. He received his PhD in distributed systems from EPFL in 2008 and his engineering degree in telecommunications from University of Belgrade in 2001. Dr. Vukolić is currently a steering committee member of Eurosys, was a PC co-chair of the SOFSEM 2011 conference, and a member of numerous program committees of major conferences. His research interests lie in the broad area of distributed algorithms and systems, including fault-tolerance, blockchain and distributed ledgers, cloud computing security and distributed storage. Postal address: IBM Research- Zurich, C379, Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland. E-mail: mvu@zurich.ibm.com