How many planet-wide leaders should there be?

Shengyun Liu
EURECOM
lius@eurecom.fr

Marko Vukolić
IBM Research - Zurich
mvu@zurich.ibm.com

ABSTRACT

Geo-replication becomes increasingly important for modern planetary scale distributed systems, yet it comes with a specific challenge: latency, bounded by the speed of light. In particular, clients of a geo-replicated system must communicate with a leader which must in turn communicate with other replicas: wrong selection of a leader may result in unnecessary round-trips across the globe. Classical protocols such as celebrated Paxos, have a *single leader* making them unsuitable for serving widely dispersed clients. To address this issue, several *all-leader* geo-replication protocols have been proposed recently, in which *every* replica acts as a leader. However, because these protocols require coordination among *all* replicas, commiting a client's request at some replica may incure the so-called "delayed commit" problem, which can introduce even a higher latency than a classical single-leader majority-based protocol such as Paxos.

In this paper, we argue that the "right" choice of the number of leaders in a geo-replication protocol depends on a given replica configuration and propose Droopy, an optimization for state machine replication protocols that explores the space between single-leader and all-leader by dynamically reconfiguring the leader set. We implement Droopy on top of Clock-RSM, a state-of-the-art all-leader protocol. Our evaluation on Amazon EC2 shows that, under typical imbalanced workloads, Droopy-enabled Clock-RSM efficiently reduces latency compared to native Clock-RSM, whereas in other cases the latency is the same as that of the native Clock-RSM.

1. INTRODUCTION

Modern internet applications are geographically distributed among datacenters, or *sites*, across the globe. To provide robust service against crashes or site outages, applications call for state machine replication (SMR) protocol, such as Paxos [6] as a basic synchronization primitive within a larger scale system (see e.g., [3]). Replication however becomes challenging at the planetary-scale due to latencies that cannot be "covered up", being bounded by the speed of light.

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

In order to reduce latency for geo-replicated applications, several *all-leader* SMR protocols have been proposed, such as [10] and [4]. In all-leader SMR, 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 request to the nearest replica, avoiding the communication with a single (and possibly remote) leader.

However, a challenge for an all-leader SMR is the coordination with distant or slow replicas, which can be the bottleneck, or even block the system. 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) is due to the need to confirm that all requests with an earlier sequence number are not "missed". For 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.

Neither single-leader nor all-leader solution fits all situations. Existing work either assumes that requests are evenly distributed among all replicas (favoring all-leader SMR), or requests are largely distributed around one replica (favoring single-leader SMR). More than often, neither of these assumptions is true in the geo-replicated context: for instance, due to time zone differences, clients located at a given site may have different access patterns at different times of a day, changing the "popularity" of sites dynamically.

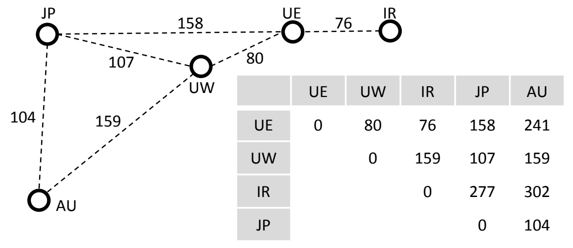
In this paper, we present Droopy, an optimization for SMR in wide area networks (WAN). Droopy explores the space between single-leader and all-leader SMR, by dynamically reconfiguring the set of leaders. Each set of leaders is selected based on previous workload and network condition. We then implement Droopy in a state-of-the-art WAN SMR protocol, namely Clock-RSM [4], and evaluate it on Amazon EC2. Our evaluation shows that, under typical imbalanced workloads, Droopy-enabled Clock-RSM efficiently reduces latency compared to native Clock-RSM, whereas in other cases the latency is the same as that of the native one.

The rest of this paper is organized as follows. In Section 2 we discuss, in the context of related work, how delayed commit problem affects the latency. In Section 3 we describe the model and assumption used in the paper. In Section 4 we give the overview and in Section 5 the details of Droopy. Finally, Section 6 depicts the way we evaluate Droopy.

2. THE "DELAYED COMMIT" PROBLEM AND STATE OF THE ART

We first describe two closely related all-leader SMR protocols: Mencius [10] and Clock-RSM [4], and how their performance is affected by the delayed commit problem. Then we briefly discuss other related work.

Mencius [10] facilitates multiple leaders by evenly partitioning sequence number space across all replicas, so that each replica can propose requests (e.g., replica 0 sequences requests 0,3,6..., replica 1 sequences 1,4,7,..., and replica 2 sequences 2,5,8,...). In case a replica lags behind, it skips the missing sequence numbers in order to catch up with other replicas. A skipping replica must however let its peers know its skips sequence numbers — leading to what is known as the "delayed commit" problem.



	UE	UW	IR	JP	AU
UE	0	80	76	158	241
UW		0	159	107	159
IR			0	277	302
JP				0	104

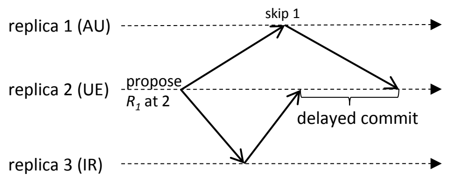
Figure 1: Round-trip latency (ms) among 5 sites on Amazon EC2 : US East (UE), US West (UW), Ireland (IR), Japan (JP) and Australia (AU).

We illustrate the delayed commit problem in Mencius by an example on Amazon EC2. The round-trip latency among 5 sites is shown in Fig. 1. The example is shown in Fig. 2(a). We assume that replica in UE (replica 2 in Fig. 2(a)) proposes request R_1 at sequence number 2, whereas replica at AU (replica 1 in Fig. 2(a)) is responsible for proposing request at sequence number 1. Because of 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 has received skip message for sequence number 1, it can execute R_1 locally. Hence, a round-trip latency between UE and AU is introduced (241 ms), which is much larger than the round-trip latency from replica 2 to a majority (76 ms) that a solution based on a single UE leader would require.

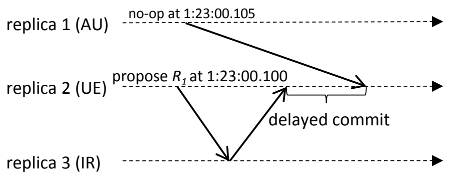
Clock-RSM [4] addresses the delayed commit problem using loosely synchronized clocks. In Clock-RSM, each replica proposes requests piggybacked with its physical clock timestamp, instead with logical sequence numbers. Requests are then ordered by associated clock timestamps. Replicas synchronize their physical clocks periodically.

However, the delayed commit problem still exists in Clock-RSM. In Fig. 2(b), after replica UE proposes R_1, replica AU sends its current clock time to all replicas. Upon UE receives the clock message from AU, it confirms that no request with earlier clock is proposed by replica AU.[1] Nevertheless, one-

[1]Because of FIFO channels assumed by Clock-RSM.



(a) Mencius



(b) Clock-RSM

Figure 2: Delayed commit problem in Mencius and Clock-RSM.

way latency from AU to UE (120 ms) is still larger than the round-trip latency from UE to a majority (76 ms).

Besides Mencius and Clock-RSM, there are several other geo-replication SMR protocols proposed recently. Some are single-leader based, such as [3]. Moreover, some protocols (e.g., [13, 7]) do not suffer from the delayed commit problem as they explore the commutativity of concurrent requests. Other protocols, such as Fast Paxos [8] allow clients to send requests directly to all replicas. In case a collision is detected, Fast Paxos relies on a single leader to re-orders the requests. Finally, some protocols (e.g., [3, 12, 2]) employ *read leases* in order to improve read performance — read leases are orthogonal to the problem we focus on here.

The Totem protocol [1] circulates a token among a set of processes to allow processes broadcasting their messages in a round-robin manner. A process can broadcast messages only upon it holds the token. Ring Paxos [11] combines ring topology of $f + 1$ processes and IP multicast in order to maximize the network throughput. These two methods are not suitable for geo-replication since clients can suffer from much extra delay (when waiting for the token or the sequential commit path of ring topology).

3. SYSTEM MODEL

We assume there are $n \geq 2f + 1$ replicas (sites), among which f can crash (but not be Byzantine) and may recover later. We allow for asynchrony in that communication time between any two replicas is not bounded — however, to circumvent the FLP impossibility [5], we assume the system to be eventually synchronous. We further assume FIFO channel between any two replicas. When referring to clock-based systems, we assume a physical clock at each replica. Clocks at different replicas are loosely synchronized by a clock synchronization protocol, such as NTP.

4. DROOPY PROTOCOL OVERVIEW

Droopy is designed to dynamically configure the set of leaders. Each set is called a *config*. Droopy splits the space of sequence indices (e.g., logical sequence numbers or physical timestamps) into ranges. Each range of indices is mapped to a config and maintained as *a lease*. In a nutshell, a lease in Droopy is a commitment of a replica to a range of indices, such that only those specific replicas, a.k.a., leaders, can propose requests. Leases are proposed by replicas and maintained by a total order primitive, which can be implemented by Paxos (we call this *L-Paxos*). In a way, Droopy follows the approach to reconfiguration proposed in [9].

When current lease is about to expire, every replica proposes a new config for the next lease, which is selected based on previous workload and network condition. In this paper we assume that the goal for Droopy is to minimize the average latency across all sites. Droopy can easily support other criteria (e.g., minimize the maximal latency for a site). Every replica proposes a new lease so that the crash of any replica will not stop the lease renewal process. However, in order to guarantee that all leases are consistent among all replicas, Droopy requires that the first delivered config for each lease is the one agreed by all. This is guaranteed by the total order primitive within L-Paxos.

It is important to notice that, although a lease can reflect a physical time range, time anomalies such as clock drifts, do not affect the correctness Droopy, but only its performance.

Generally, a client submits a request by contacting the nearest replica, that we call the *source replica* of the request. The source replica then proposes the request if it is one of the leaders in the current lease. Otherwise, the source replica propagates the request to the 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. Upon a request is committed locally, each replica executes the request, and the source replica further replies to the client. Besides, each updates its *frequency array* monitor, that records the number of requests from each source replica.

5. PROTOCOL DETAILS

In this section we explain the detail of Droopy. The variable definitions and pseudocode is given in Alg. 1.

Proposing. Client c first sends req to the nearest replica s_i and waits for reply from s_i. Upon arrival of req from c (line 1, Alg. 1), s_i becomes the source replica for req (line 2). Then, s_i obtains and updates the sequence index by getOrder() from underlying SMR protocol, e.g., Mencius or Clock-RSM (line 3). Then, 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 sequence index 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).

Committing. In the commit phase, Droopy needs to modify the underlying SMR protocol at the condition which checks whether there might be unreceived requests preceding sequence index sn. In particular, this check in all-leader protocols involves all replicas (leaders), whereas in Paxos it involves a single leader. In Droopy, this condition depends on which replicas are leaders in each lease. The modification is shown as function updated(sn).

Upon req is committed at replica s_i (line 10), i.e., updated(sn) is true, all SMR protocols maintain the invariant that req is replicated by a majority of replicas and all preceding requests are committed locally at s_i. Then, 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 to expire (in λ sequence indices, lines 15-18),

Algorithm 1 Dynamic reconfiguration at replica s_i.

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

```
1:  upon receive ⟨Request, $req, s_j$⟩ from client $c$ or $s_k$
2:      $src \leftarrow s_j = null\ ?\ s_i : s_j$                /* source replica */
3:      $sn \leftarrow$ getOrder()
4:      while $sn \geq LE_{ln}$ do                      /* update lease */
5:          $ln \leftarrow ln + 1$
6:      if $s_i \in config_{ln}$ then                          /* leader */
7:          propose($req, sn, src$) in underlying SMR protocol
8:      else                                             /* non-leader */
9:          sends ⟨Request, $req, src$⟩ to $p_i \in config_{ln}$

10: upon decide($req, sn, src$) in underlying SMR protocol, in
    which updated($sn$) is applied and $true$ (see lines 23-24)
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 ⟨Reply, $rep$⟩ 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
20:     if $config_{ln'} = null$ then /* 1st decision for lease $ln'$ */
21:         $config_{ln'} = config$
22:         $LE_{ln'} = LE$

23: function updated($sn$)                       /* returns boolean */
24:     return $\forall s_k$ and $ln$ : if $s_k \in config_{ln}$ then
    $latest[s_k] \geq LE_{ln}$ or $sn \leq latest[s_k]$

25: abstract function getOrder()
26: abstract function getNewConfig($freq, d$)
```

replica s_i first selects a new config $config$ based on array f_* and table $d_{*,*}$; then s_i proposes $config$ piggybacked with lease number $ln + 1$ in L-Paxos and resets the frequency array $freq_*$. Upon $config$ in lease ln' is decided at replica s_i (lines 19-22), if $config$ is the first decision for ln', then s_i updates the leader set $config_{ln'}$ and the end index $LE_{ln'}$.

Config selection. To select a suitable config, replica s_i enumerates all possible combinations of sets of leaders. Given a set of leaders, s_i calculates the estimated latency based on latency table $d_{*,*}$ and frequency array f_*.

The calculation of estimated latency depends on how underlying SMR protocol proceeds. For example, to confirm that all previous requests have been received, one-way latency from the farthest replica matters in Clock-RSM; whereas, this latency for Mencius can be a round-trip.

More specifically, the latency for req is dominated by three conditions : ① the time it takes for req to be replicated by a majority; ② the time it takes replica s_j to confirm that no request with a sequence index smaller than that of req will be proposed by some leader; and ③ the time it takes for

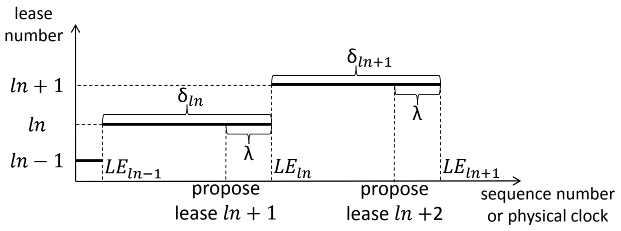


Figure 3: Lease renewal.

s_i to commit all requests that precede req. Based on these three conditions, the calculation for each SMR protocol can be designed separately. As an illustration, the calculation for Clock-RSM is shown in Alg. 2 in line 4.

Algorithm 2 Clock-RSM function.

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

3: function getNewConfig($freq, d$)
4:     return $s \subseteq replicas$ s.t.
```

$$min(\sum_{i=1}^{n} freq_i \times min(max \left\{ \begin{array}{c} d_{i,j} + median(d_{j,k} + d_{k,i} | \forall s_k), ① \\ d_{i,j} + max(d_{k,i} | \forall s_k \in s), ② \\ d_{i,j} + max((median(d_{k,l} + d_{l,i}) | \forall s_l) | \forall s_k \in s), ③ \end{array} \right\} | \forall s_j \in s) | \forall s \subseteq replicas)$$

6. EVALUATION

Implementation We implement Droopy and Clock-RSM in Java and deploy them on Amazon EC2. Both replicas and clients are deployed in c3.large instances that run Ubuntu Server 14.04. We use TCP as the transport protocol to guarantee FIFO channels. Replicas run NTP daemon to keep the clock synchronized. The timestamp is obtained by $System.currentTimeMillis()$. We set δ to 10 seconds and λ to 2 seconds for Droopy.

Static workload We evaluate the protocols under imbalanced workload. In each experiment, 40 clients at one specific site issue requests of 64B to their source replica. The results for $f = 1$ and $f = 2$ are shown in Fig. 4. The latency among replicas are shown in Fig. 1.

Fig. 4 shows that Droopy can efficiently reduce the latency when workload deployed at UE and IR in Fig. 4(a), or UE, IR and JP in Fig. 4(b). This is because at these replicas the coordination time in Clock-RSM is larger than the round-trip latency from a majority. In other cases, the performance of Droopy is the same as that of native one.

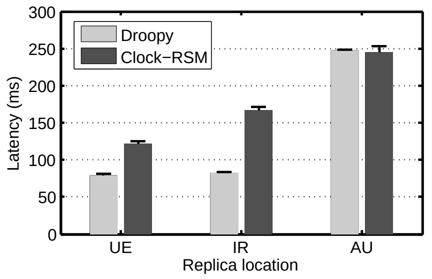
7. ACKNOWLEDGEMENTS

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).

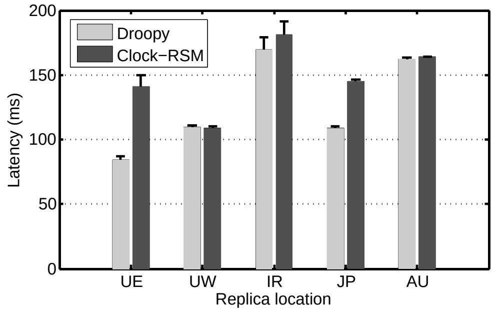
Shengyun Liu's work is also supported by the China Scholarship Council.

8. REFERENCES

[1] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342, Nov. 1995.
[2] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *PODC*, 2007.
[3] J. C. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.
[4] 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 *DSN*, 2014.
[5] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. 32(2):374–382, Apr. 1985.
[6] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998.
[7] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.
[8] L. Lamport. Fast paxos. *Distributed Computing*, 19:79–103, 2006. 10.1007/s00446-006-0005-x.
[9] L. Lamport, D. Malkhi, and L. Zhou. Vertical paxos and primary-backup replication. In *PODC*, 2009.
[10] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for wans. In *OSDI*, 2008.
[11] P. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *DSN, 2010*, pages 527–536, June 2010.
[12] I. Moraru, D. G. Andersen, and M. Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *SoCC*, 2014.
[13] I. Moraru et al. There is more consensus in egalitarian parliaments. In *SOSP*, 2013.



(a) $f = 1$



(b) $f = 2$

Figure 4: Average latency (bars) and 95%ile (lines atop bars) of imbalanced workload.