Clock-RSM: Low-Latency Inter-Datacenter State Machine Replication Using Loosely Synchronized Physical Clocks (Technical Report)

Jiaqing Du*, Daniele Sciascia†, Sameh Elnikety‡, Willy Zwaenepoel*, and Fernando Pedone†
*EPFL, Lausanne, Switzerland
†University of Lugano (USI), Lugano, Switzerland
‡Microsoft Research, Redmond, WA, USA

***Abstract*—This paper proposes Clock-RSM, a new state machine replication protocol that uses loosely synchronized physical clocks to totally order commands for geo-replicated services. Clock-RSM assumes realistic non-uniform latencies among replicas located at different data centers. It provides low-latency linearizable replication by overlapping 1) logging a command at a majority of replicas, 2) determining the stable order of the command from the farthest replica, and 3) notifying the commit of the command to all replicas. We evaluate Clock-RSM analytically and derive the expected command replication latency. We also evaluate the protocol experimentally using a geo-replicated key-value store deployed across multiple Amazon EC2 data centers.**

I. Introduction

Many online services replicate their data at multiple geographic locations to improve locality and availability [2, 5, 6]. User requests can be served at nearby replicas, thus reducing latency. By deploying replicas at multiple data centers, a system can tolerate the failure of some replicas due to server, network, and data center outage.

The state machine approach [19] is often used to replicate services consistently. All replicas execute the same set of commands in the same order deterministically. If replicas start from the same initial state, their states are consistent after executing the same sequence of commands.

Although many protocols have been proposed to implement the state machine approach, they are not well suited for geo-replicated systems because of high replication latency. The latency of a protocol in this environment is dominated by message transmission, rather than message processing and logging. A round trip between two data centers can take hundreds of milliseconds. In contrast, message processing and logging to stable storage takes much less time, from microseconds to a few milliseconds. In addition, the latencies among data centers are not uniform and vary depending on the physical distance as well as the wide-area network infrastructure.

Multi-Paxos [12, 13] is one of the most widely used state machine replication protocols. One replica is designated as the leader, which orders commands by contacting a majority of replicas in one round trip of messages. A non-leader replica forwards its command to the leader and later receives the commit notification, adding the latency of another round trip, which is significant in a wide-area setting. Mencius [16] addresses this problem by avoiding a single leader. It rotates the leader role among the replicas according to a predefined order. Committing a command in Mencius takes one round trip of messages. However, Mencius suffers from the *delayed commit* problem: A command can be delayed by another concurrent command from a different replica by up to one round-trip latency. In addition, in some cases a replica needs to contact the farthest replica to commit a command while Multi-Paxos only requires the leader plus a majority. With realistic non-uniform inter-data center latencies, Mencius provides higher latency than Multi-Paxos in many cases.

We propose Clock-RSM, a state machine replication protocol that provides low latency for consistent replication across data centers. Clock-RSM uses loosely synchronized physical clocks at each replica to totally order commands. It avoids both the single leader required in Multi-Paxos and the delayed commit problem in Mencius. Clock-RSM requires three steps to commit a command: 1) logging the command at a majority of replicas, 2) determining the stable order of the command from the farthest replica, and 3) notifying the commit of the command to all replicas. It overlaps these steps to reduce replication latency. For many real world replica placements across data centers, Clock-RSM needs only one round-trip latency to a majority of replicas to commit a command. As Clock-RSM does not mask replica failures as in Paxos, we also introduce a reconfiguration protocol that removes failed replicas and reintegrates recovered replicas to the system.

We evaluate Clock-RSM analytically and experimentally, and compare it with Paxos-bcast and Mencius-bcast, variants of Multi-Paxos and Mencius with latency optimizations. We derive latency equations for these protocols. We also implement these protocols in a replicated key-value store and deploy it arcoss three and five Amazon EC2 data centers. We find that Clock-RSM has lower latency than Paxos-bcast at the non-leader replicas and similar or slightly higher latency at the leader replicas. In addition, Clock-RSM always provides lower latency than Mencius-bcast.

The key contributions of this paper are the following:

- We describe Clock-RSM, a state machine replication protocol using loosely synchronized physical clocks (Section III).
- We derive the commit latency of Clock-RSM, Multi-Paxos, Paxos-bcast, and Mencius-bcast analytically assuming non-uniform latencies (Section IV).
- We present a reconfiguration protocol for Clock-RSM that automatically removes failed replicas and reintegrates recovered ones (Section V).
- We evaluate Clock-RSM on Amazon EC2 and demonstrate its latency benefits by comparison with other protocols (Section VI).

II. Model and Definitions

In this section we describe our system model and define the properties that Clock-RSM guarantees.

A. System Model

We assume a standard asynchronous system that consists of a set of interconnected processes (clients and replicas). Processes communicate through message passing. We assume that messages are eventually delivered by their receivers, and that there is no bound on the time to deliver a message. To simplify the presentation of the protocols, we assume that messages are delivered in FIFO order.

Processes may fail by crashing and can later recover. We assume no byzantine failures. Processes have access to stable storage, which survives failures. Processes are equipped with a failure detector. We use failure detectors to ensure liveness. Failure detectors may provide wrong results, but eventually all faulty processes are suspected and at least one non-faulty process is not suspected. In practice, such a failure detector can be implemented by timeouts.

We assume the server where a replica runs is equipped with a physical clock. Clocks are loosely synchronized by a time synchronization protocol, such as NTP [1]. A clock provides monotonically increasing timestamps. The correctness of Clock-RSM does not depend on the synchronization precision.

B. State Machine Replication

State machine replication is a technique for implementing fault-tolerant services [19]. It replicates a state machine over a set of replicas. A state machine consists of a set of commands that may read and/or write the current state and produce an output. Replicas coordinate to execute commands issued by clients to achieve the desired consistency criteria.

Clients observe *linearizable* executions [7]. An execution σ is linearizable if there exists a permutation of all commands in σ such that: 1) it respects the semantics of the commands, as defined in their sequential specification; and 2) it respects the real-time ordering of commands across all clients. Linearizability can be achieved by ensuring that each replica executes commands in the *same order*. This, together with the assumption that commands are deterministic and executed atomically, ensures that replicas transit through the same states and produce the same output for each command.

Symbols	Definitions
$Spec$	all replicas, active or failed, in the system specified by the system administrator
$Config$	current configuration that includes all active replicas in $Spec$
Log	command log on stable storage
$Clock$	latest time of the physical clock
$PendingCmds$	commands pending to commit
$LatestTV$	latest clock timestamps from all replicas of $Config$, $\|LatestTV\| = \|Config\|$
$RepCounter$	command replication counter

Table I Definition of symbols used in Algorithm 1.

C. Geo-Replication

For a geo-replicated service, such as a data store, each replica is placed in a distinct data center. Users issue requests to their nearest data center, and the requests are handled by application servers, which contact the local replica of the service. Hence, from this point of view, clients are the application servers and they are local (within the same data center) to a replica of the geo-replicated service.

III. Clock-RSM

In this section we describe Clock-RSM in detail.

Clock-RSM is a multi-leader protocol. A client connects to its nearby replica within the same data center, and each replica coordinates commands of its own clients. A replica assigns a unique timestamp to a client command, broadcasts it, and waits for the acknowledgements broadcast by other replicas once logging it on their stable storage. Every replica executes commands serially in the timestamp order after they are committed. A replica knows that a command has committed if all the following three conditions hold:

1) **Majority replication.** A majority of replicas have logged the command;
2) **Stable order.** The replica has received all commands with a smaller timestamp;
3) **Prefix replication.** All commands with a smaller timestamp have been replicated by a majority.

Algorithm 1 gives the pseudocode of the Clock-RSM replication protocol. Table I defines the symbols used in the protocol.

A. Protocol States

Each replica maintains three hard states: 1) $Spec$, the specification of all replicas in the system; 2) $Config$, the current configuration that includes all active replicas in $Spec$, $Config \subseteq Spec$; 3) Log, the command log.

The system administrator specifies $Spec$ before the system starts, and we assume the specification of replicas is fixed during the life time of the system. Clock-RSM requires a

Algorithm 1 Replication Protocol at Replica r_m

```
1: upon receive ⟨REQUEST cmd⟩ from client
2:     ts ← Clock
3:     send ⟨PREPARE cmd, ts⟩ to replicas in Config

4: upon receive ⟨PREPARE cmd, ts⟩ from r_k
5:     PendingCmds ← PendingCmds ∪ {⟨cmd, ts, k⟩}
6:     LatestTV[k] ← ts
7:     append ⟨PREPARE cmd, ts⟩ to Log
8:     wait until ts < Clock
9:     clockTs ← Clock
10:    send ⟨PREPAREOK ts, clockTs⟩ to replicas in Config

11: upon receive ⟨PREPAREOK ts, clockTs⟩ from r_k
12:    LatestTV[k] ← clockTs
13:    RepCounter[ts] ← RepCounter[ts] + 1

14: upon ∃⟨cmd, ts, k⟩ ∈ PendingCmds, s.t. COMMITTED(ts)
15:    append ⟨COMMIT ts⟩ to Log
16:    result ← execute cmd
17:    if k = m then
18:        send ⟨REPLY result⟩ to client
19:    remove ts from PendingCmds, RepCounter

20: function COMMITTED(ts)
21:    return RepCounter[ts] ≥ ⌊|Spec|/2⌋ + 1 ∧
22:           ts ≤ min(LatestTV) ∧
23:           ∄ts' ∈ PendingCmds, s.t. ts' < ts
```

majority of replicas in the specification to be non-faulty, which means $Config$ should contain at least a majority subset of $Spec$. The failure or recovery of a replica triggers changes in $Config$. A reconfiguration protocol removes failed replicas from and adds recovered replicas to $Config$ (Section V). Without loss of generality, our explanation and analysis of Clock-RSM assume that a failed replica recovers and joins the replication fast, i.e., $Spec = Config$.

Each replica also maintains some soft states when executing the protocol: 1) $PendingCmds$, a set containing timestamps of commands that have not been committed yet; 2) $RepCounter$, a dictionary that stores the number of replicas that have logged a command; 3) $LatestTV$, a vector of timestamps with the same size as $Config$. $LatestTV[k]$, the kth element of $LatestTV$, contains the latest known timestamp from replica r_k. It indicates that all commands originated from r_k with timestamp smaller than $LatestTV[k]$ have been received.

B. Protocol Execution

Clock-RSM is given in Algorithm 1. We explain how it totally orders and executes client commands.

1. When a replica receives $\langle$REQUEST $cmd\rangle$ from a client, where cmd is the requested command to execute, it assigns to the command its latest clock time. We call this replica the *originating* replica of the command. The replica then sends a prepare message, $\langle$PREPARE $cmd, ts\rangle$, to all replicas to replicate cmd. ts is the assigned timestamp that uniquely identifies and orders cmd. Ties are resolved by using the id of the command's originating replica. (lines 1-3)

Algorithm 2 Periodic clock time broadcast at replica r_m

```
1: upon Clock ≥ LatestTV[m] + Δ
2:     ts ← Clock
3:     send ⟨CLOCKTIME ts⟩ to all replicas in Config

4: upon receive ⟨CLOCKTIME ts⟩ from replica r_k
5:     Latest[k] ← ts
```

2. When a replica receives $\langle$PREPARE $cmd, ts\rangle$, the logging request, from replica r_k, it adds the command to $PendingCmds$, the set of pending commands not committed yet. It updates the kth element of $LatestTV$ with ts, reflecting the latest time it knows of r_k. (lines 4-6)

It appends the message to its log on stable storage and acknowledges that it has logged the command with $\langle$PREPAREOK $ts, clockTs\rangle$, where $clockTs$ is the replica's clock time. Notice that before acknowledging the command, the replica waits until its local clock time is greater than the timestamp of the command. That is, it promises not to send any message with a timestamp smaller than ts afterwards. To reduce the total commit latency, the acknowledgment is sent to all replicas. (lines 7-10)

The wait (at line 8) is highly unlikely with reasonably synchronized clocks, which normally provide much smaller clock skew than one-way message latency between data centers. Replicas send both PREPARE and PREPAREOK messages in timestamp order (ts in PREPARE and $clockTs$ in PREPAREOK). This guarantees replicas send monotonically increasing timestamps carried by the two types of messages.

3. When a replica receives $\langle$PREPAREOK $ts, clockTs\rangle$ from replica r_k, it learns its latest timestamp and updates $LatestTV$ with $clockTs$ accordingly. The replica then increments the replication counter $RepCounter[ts]$ to record the number of replicas that have logged the command with ts. $RepCounter[ts]$ has a default value of 0. (lines 11-13)

4. A replica knows that a command has committed when the following three conditions hold: 1) It receives replication acknowledgements from a majority of replicas. 2) It will not receive any message with a smaller timestamp from any replica. 3) It has executed all commands with a smaller timestamp. (lines 20-23)

When a replica learns the commit of a command with timestamp ts, it appends $\langle$COMMIT $ts\rangle$, the *commit mark*, to its log and executes the command. Commit marks are appended to the log in timestamp order. This helps a replica replay the log in the correct order during recovery, as we show in Section V. If the command is from one of the replica's clients, it sends the execution result back to the client. Finally, the replica removes the command from $PendingCmds$ and $RepCounter$. (lines 14-19)

We prove Clock-RSM given in Algorithm 1 and its reconfiguration protocol given in Algorithm 3 in Section V in the appendix.

C. Extension

We present an extension to Algorithm 1 that further improves its latency. Algorithm 2 gives the pseudocode. Each replica periodically broadcasts its latest clock time if it does not receive frequent enough client requests. Δ is the minimum interval at which a replica broadcasts its latest clock time. If there are frequent enough client requests, a replica does not need to broadcast CLOCKTIME messages. When a replica receives a CLOCKTIME message from replica r_k, it updates $LatestTV[k]$ accordingly. This extension requires a replica to send PREPARE, PREPAREOK, and CLOCKTIME messages in timestamp order.

This extension improves latency *only* in one case: Among all replicas, only one replica serves very infrequent client requests while the other replicas do not serve client requests at all. We explain why this is the case in Section IV. Notice that this extension makes Clock-RSM non-quiescent, although it improves latency.

IV. LATENCY ANALYSIS

In this section we analyze the latency of Clock-RSM (Algorithm 1) and its extension (Algorithm 2). We also compare Clock-RSM with Paxos and Mencius analytically.

We assume N replicas deployed in different data centers, and denote the set of all replicas by R, which is $\{r_k \mid 0 \leq k \leq N-1\}$. We assume that latencies between replicas are non-uniform, and define $d(r_i, r_j)$ as the one-way message latency between replica r_i and r_j. We assume symmetric network latency between two replicas: $d(r_i, r_j) = d(r_j, r_i)$. Given high network latencies in a WAN, we ignore the latency introduced by local computation and disk I/O, as well as clock skew.

A. Clock-RSM

Assume that r_i is the originating replica of command cmd and assigns timestamp ts to it. As described in Section III, a replica in Clock-RSM knows a command committed if three conditions hold. We analyze the latency requirement of each condition and derive the latency required to commit cmd at r_i below.

1) Majority replication requires cmd to be logged by a majority of replicas. To satisfy this condition, r_i needs to receive PREPAREOKs for cmd from a majority. The required latency is one round trip from r_i to a majority: $2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\})$. We denote lc_1 the latency required to satisfy majority replication.

2) Stable order requires that cmd has the smallest timestamp among all commands that have not committed.

In the worst case, if no replica sends a message to r_i between the time that r_i assigns cmd a timestamp and cmd is logged at all replicas, r_i has to rely on the PREPAREOKs of cmd from all replicas to determine its stable order. The latency is $2 * max(\{d(r_i, r_k) \mid \forall r_k \in R\})$. We denote this latency by lc_2^{worst}.

In the best case, when r_i assigns cmd a timestamp, around the same time, if every replica sends a message with a timestamp greater than ts to r_i, then r_i knows that cmd is stable once all these messages arrive at r_i. The message can be either PREPARE or PREPAREOK. The latency is $max(\{d(r_i, r_k) \mid \forall r_k \in R\})$. We denote this latency by lc_2^{best}.

If the extension in Algorithm 2 is enabled, replicas broadcast their clock time every Δ time units. Therefore r_i determines the stable order of cmd after $lc_2^{best} + \Delta$ time units, regardless of commands being submitted concurrently. In practice, we expect Δ to be a small value. Hence lc_2^{worst} is roughly the same as lc_2^{best} with this extension enabled.

3) Prefix replication is satisfied when all commands with a smaller timestamp than ts are replicated by a majority.

In the worst case, when r_i assigns cmd a timestamp, around the same time, if every other replica also assigns to its own command a slightly smaller timestamp than ts, then r_i needs to know that all of these commands are replicated by a majority, after the command becomes stable. That is, for each of these commands, r_i waits for its PREPAREOK message from a majority. The latency is $max(\{median(\{d(r_j, r_k) + d(r_k, r_i) \mid \forall r_k \in R\}) \mid \forall r_j \in R\})$. We denote this latency by lc_3^{worst}.

In the best case, when cmd becomes stable at r_i, if all commands with a smaller timestamp than ts have committed, this condition holds immediately. Hence it is dominated by the previous two conditions, and its latency can be ignored when computing the final commit latency. We denote by lc_3^{best} the latency in this case. We explain how this case happens later in the section.

We now derive the overall latency of committing a command under two different workloads.

Balanced workloads. Each replica serves client requests at moderate or heavy load. That is, every replica sends and receives PREPARE and PREPAREOK frequently.

This is the best case for condition 2 and the worst case for condition 3. For r_i, the latency of committing a command is $max(lc_1, lc_2^{best}, lc_3^{worst}) = max(2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{median(\{d(r_j, r_k) + d(r_k, r_i) \mid \forall r_k \in R\}) \mid \forall r_j \in R\}))$.

Imbalanced workloads. Only one replica serves client requests. If the workload is moderate or heavy, the replica sends PREPARE messages frequently. Since every replica broadcasts PREPAREOK, these messages of previous commands carry back the latest clock time of other replicas and help reduce the stable order duration of the current one. This is the best case for condition 2. As only one replica proposes commands, when the replica knows the current command is replicated by a majority and is stable, all previous commands must have committed. Hence it is also the best case for condition 3. The latency is $max(lc_1, lc_2^{best}, lc_3^{best}) = max(2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{d(r_i, r_k) \mid$

$\forall r_k \in R\}))$.

If the workload is light and the replica sends PREPARE messages infrequently, the PREPAREOK messages of previous commands do not help the stable order condition any more. This is the worst case for condition 2 and the best case for condition 3. The latency is $max(lc_1, lc_2^{worst}, lc_3^{best})$ $= 2 * max(\{d(r_i, r_k) \mid \forall r_k \in R\})$. With the extension in Algorithm 2 enabled, lc_2^{worst} is roughly the same as lc_2^{best}. Hence the latency becomes $max(lc_1, lc_2^{best} + \Delta, lc_3^{best}) = max(2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{d(r_i, r_k) \mid \forall r_k \in R\}) + \Delta)$. This is the only case where enabling the extension given in Algorithm 2 helps latency.

In summary, the latency of Clock-RSM depends on the locations of the replicas and the workloads. lc_1 is the round-trip latency between the originating replica and a majority of all replicas. lc_2 is bounded by the maximum one-way latency between the originating replica and other replicas. lc_3 is bounded by the maximum two-hop latency between the originating replica and other replicas via a majority. The message complexity of Clock-RSM is $\mathcal{O}(N^2)$ as every replica broadcasts PREPAREOK messages.

B. Paxos

Multi-Paxos [12, 13] is the most widely used Paxos variant. We use Paxos to refer to Multi-Paxos in the rest of the paper. With Paxos, one replica is designated as the leader, which coordinates replication and totally orders commands. We denote the leader in Paxos by r_l .

A non-leader replica r_i in Paxos experiences the following latency to commit a command. r_i needs $d(r_i, r_l)$ time to forward the command it proposes to leader r_l. r_l sends phase 2a messages to all replicas. All replicas reply to the leader with phase 2b messages. In order for r_l to learn that a command is committed, it waits for phase 2b messages from a majority. This takes $2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$ time. Finally, the leader needs $d(r_l, r_i)$ time to notify r_i the commit of its command. Thus, the overall latency is $2 * d(r_i, r_l) + 2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$. If the leader proposes a command, the latency reduces to $2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$.

A well-known optimization allows Paxos replicas to broadcast phase 2b messages, thus saving the last message from the leader to the originating replica of a command. Replicas learn the outcome of a command without the assistance of the leader. In this case, r_i waits for phase 2b messages from a majority. The overall latency is $d(r_i, r_l) + median(\{d(r_l, r_k) + d(r_k, r_i) \mid \forall r_k \in R\})$. For the leader replica, the latency is still $2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$. We use *Paxos-bcast* to refer to the Paxos variant that broadcasts its phase 2b message.

Although Paxos-bcast improves latency, it increases the message complexity of Paxos from $\mathcal{O}(N)$ to $\mathcal{O}(N^2)$.

C. Mencius

Mencius [16] rotates the leader role among all the replicas based on a predefined order. Each replica serves client requests at the rounds it coordinates. Mencius can also save the last step message, which is used to notify the commit of a command, by broadcasting the replication acknowledgement message. We use *Mencius-bcast* to refer to Mencius with this latency optimization.

Under imbalanced workloads when only one replica proposes commands, regardless of light or heavy load, Mencius-bcast always needs one round-trip message from the replica to *all* replicas to commit a command. Hence it takes $2 * max(\{d(r_i, r_k) \mid \forall r_k \in R\})$ time to commit a command at replica r_i.

Under balanced workloads, due to the delayed commit problem, the commit latency at r_i is between q and $q + max(\{d(r_i, r_k) \mid \forall r_k \in R\})$, where q is the latency of Clock-RSM with the same workloads. Clock-RSM does not suffer from the delayed commit problem because it uses physical clocks to assign command timestamps[1].

Therefore, compared with Clock-RSM, Mencius-bcast always requires higher latency or at most the same in "lucky" cases. Similar to Paxos-bcast, Mencius-bcast increases the message complexity of Mencius from $\mathcal{O}(N)$ to $\mathcal{O}(N^2)$.

D. Intuition and Comparison

We summarize our latency analysis of the four protocols in Table II, and compare Clock-RSM and Paxos-bcast below. We do not discuss Mencius-bcast and Paxos because they have higher latency than Clock-RSM and Paxos-bcast, respectively.

At non-leader replicas, Paxos-bcast requires more message steps than Clock-RSM. If we assume that the latencies between any two replicas are the same, Clock-RSM provides lower latency. In practice latencies among data centers are not uniform. Simply counting message steps is not sufficient to determine how a protocol performs.

For both protocols, a replica needs to log a command at a majority. The replica that sends the logging requests, the replica that receives the logging confirmations, and the majority replicas that log the commands may be different. However, communicating with a majority means median latency, which avoids the paths of high latency between far away replicas. For Clock-RSM, the prefix replication condition also requires a majority and it overlaps with majority replication, hence it does not increase the overall commit latency much. As a result, as long as replicas are not too far apart, such as one replica is far away from the rest, logging a command at a majority replicas does not differ much in these two protocols.

The two protocols differ mainly in command ordering. For Paxos-bcast, a non-leader replica needs one additional

[1]This can be easily verified in Figure 3 in [16].

Protocol	Steps	Complexity	Latency
Paxos	4 / 2	$\mathcal{O}(N)$	Leader: $2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$ Non-leader: $2 * d(r_i, r_l) + 2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$
Paxos-bcast	3 / 2	$\mathcal{O}(N^2)$	Leader: $2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$ Non-leader: $d(r_i, r_l) + 2 * median(\{d(r_l, r_k) \mid \forall r_k \in R\})$
Mencius-bcast	2	$\mathcal{O}(N^2)$	Imbalanced: $2 * max(\{d(r_i, r_k) \mid \forall r_k \in R\})$ Balanced: $[q, q + max(\{d(r_i, r_k) \mid \forall r_k \in R\})]$, q is latency of Clock-RSM
Clock-RSM	2	$\mathcal{O}(N^2)$	Imbalanced: $max(2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{d(r_i, r_k) \mid \forall r_k \in R\}))$ Balanced: $max(2 * median(\{d(r_i, r_k) \mid \forall r_k \in R\}), max(\{d(r_i, r_k) \mid \forall r_k \in R\}),$ $max(\{median(\{d(r_j, r_k) + d(r_k, r_i) \mid \forall r_k \in R\}) \mid \forall r_j \in R\}))$

Table II Number of message steps, message complexity, and command commit latency of Paxos, Paxos-bcast, Mencius-bcast, and Clock-RSM.

message to forward a command to the leader. We denote the forwarding latency by d_{fwd}. For Clock-RSM, the originating replica of a command requires a message from every other replica to determine the stable order of the command. Only a message with a greater timestamp than the command timestamp helps the stable order process. In the best case, this message is sent out roughly at the same time when the command is sent out for majority replication. Receiving the message from the farthest replica, taking $d_{max} = lc_2^{best}$ time, overlaps with the round-trip latency of majority replication, taking $2 * d_{median} = lc_1$ time. Hence, $d_{max} - 2 * d_{median}$ is the latency introduced by command ordering in Clock-RSM.

Combining the above analysis, Clock-RSM provides lower latency than Paxos-bcast at a non-leader replica as long as $d_{max} - 2 * d_{median} < d_{fwd}$, which means the highest latency is smaller than the sum of twice median latency and one forwarding latency. This condition is not demanding. Our measurements of latencies among Amazon EC2 data centers in Section VI show that it holds in most cases. At the leader replica, Clock-RSM provides the same latency if $d_{max} - 2 * d_{median} \leq 0$, which means the highest latency is smaller than or equal to twice of median latency. In this case, majority replication dominates the overall commit latency. Our measurements show that this condition does not hold all the time. However, even when it does not hold, d_{max} is not much greater than $2 * d_{median}$, meaning that Clock-RSM does not lose much in those cases.

In summary, Clock-RSM provides lower latency than Paxos-bcast under conditions that are easy to satisfy in practice. In general, the more uniform the latencies among replicas are, the more likely Clock-RSM provides lower latency.

V. Failure Handling

To order commands, replicas rely on knowing the latest clock time of every replica in the current configuration. As a consequence, Clock-RSM may stall in case of failure of a replica or network partitions in the current configuration. In this section, we describe a reconfiguration protocol for Clock-RSM, which removes failed replicas and reintegrates recovered ones.

A. Reconfiguration

The reconfiguration protocol is given in Algorithm 3. It exposes a RECONFIGURE function that is triggered when a failure detector suspects that a replica has failed, or a recovered replica asks to rejoin. RECONFIGURE takes a new configuration as the argument, which specifies the new membership of the system after reconfiguration. The protocol uses primitives PROPOSE(k, m_p) and DECIDE(k, m_d) of consensus. In practice one can use a protocol like Paxos [12, 13] to implement the primitives. A replica proposes value m_p as kth consensus instance. Eventually, all correct processes decide on the same final value m_d, among the ones proposed. We use consensus because two or more replicas may trigger RECONFIGURE with a different configuration. The protocol also introduces a new hard state: a monotonically increasing $Epoch$ number. $Epoch$ is initially 0, and is incremented after each reconfiguration. This allows us to ignore messages from older epochs, issued by replicas which have not reconfigured yet. Notice that we do not assume $Config = Spec$ in this section, because reconfiguration changes $Config$. The protocol works as follows.

1. A replica r_k that triggers RECONFIGURE sends a $\langle$SUSPEND $e, cts\rangle$ message to all replicas in $Spec$. e is the next epoch number and cts is the timestamp of the last commit mark in r_k's Log. r_k then waits for SUSPENDOK replies from a majority in $Spec$. The purpose of this phase is two-fold. First, replicas that receive a SUSPEND message stop handling PREPARE requests from other replicas and REQUEST messages from clients, essentially freezing their logs. Second, r_k collects all logged commands with a timestamp greater than cts, from a majority in $Spec$. This includes all commands that could have been committed by a failed replica. Finally, r_k invokes the eth consensus instance over all replicas in $Spec$ by proposing $config_{new}$, the next configuration to use, timestamp cts, and the above set of commands.

2. Eventually all non-faulty replicas learn about the decision for the eth consensus instance. The decision includes enough information to start a new epoch such that all replicas in $config_{new}$ start from the same state. To do so, replicas

Algorithm 3 Reconfiguration protocol at Replica r_m

```
1: function RECONFIGURE(config_new)
2:     e ← Epoch + 1
3:     cts ← timestamp of the last commit mark in Log
4:     send ⟨SUSPEND e, cts⟩ to all replicas in Spec
5:     wait for ⟨SUSPENDOK e, cmds_k⟩ from a majority of Spec
6:     PROPOSE(e, config_new, cts, ⋃_k cmds_k)
7: upon receive ⟨SUSPEND e, cts⟩ from r_k
8:     stop processing REQUEST and PREPARE messages
9:     cmds ← {∀⟨cmd, ts⟩ ∈ Log | ts > cts}
10:    send ⟨SUSPENDOK e, cmds⟩ to r_k
11: upon DECIDE(e, config_new, ts, cmds)
12:    cts ← timestamp of the last commit mark in Log
13:    if ts > cts then
14:        cmds ← cmds ∪ STATETRANSFER(cts, ts)
15:    remove all ⟨PREPARE c, t⟩ from Log, s.t. t > ts and c is
       not executed yet
16:    for all ⟨cmd, ts⟩ ∈ cmds do                ▷ in order of ts
17:        if ⟨PREPARE cmd, ts, k⟩ ∉ Log then
18:            append ⟨PREPARE cmd, ts⟩ to Log
19:            append ⟨COMMIT ts⟩ to Log
20:            execute cmd
21:    Epoch ← e
22:    Config ← config_new
23:    resize and update LatestTV
24:    resume processing REQUEST and PREPARE messages
25: function STATETRANSFER(from, to)
26:    send ⟨RETRIEVECMDS from, to⟩ to all replicas in Spec
27:    wait for ⟨RETRIEVEREPLY cmds_k⟩ from majority of Spec
28:    return ⋃_k cmds_k
29: upon receive ⟨RETRIEVECMDS from, to⟩ from r_k
30:    cmds ← {∀⟨cmd, ts⟩ ∈ Log | from < ts ≤ to}
31:    send ⟨RETRIEVEREPLY cmds⟩ to r_k
```

remove all entries with timestamp greater than ts from their Log, where ts is the timestamp in the consensus decision and apply all commands that could have been committed in timestamp order. Replicas then install the new epoch number and configuration. They also resize $LatestTV$ based on the new configuration and update its elements. Finally, the normal case replication protocol can resume. Notice that some replicas may lag behind when the last commit mark in their Log is smaller than the decided timestamp. In such a case, a replica initiates a state transfer to fetch all commands up to ts, before applying the commands decided by consensus.

B. Recovery and reintegration

We next discuss how a replica recovers from its Log and is reintegrated to the existing active replicas. A replica may fail and recover in a short time, without triggering the reconfiguration process. The $Epoch$ number is used to determine whether a reconfiguration has meantime happened or not. Log is used to replay the history of commands up to the point in which the replica failed. Recall that log entries are of two types in Clock-RSM, either PREPARE or COMMIT. A PREPARE entry contains a command with the timestamp, but they do not necessarily appear in timestamp order in Log. A COMMIT entry contains a timestamp only and is logged in timestamp order. In addition, Clock-RSM guarantees that a COMMIT entry is appended to Log after the corresponding PREPARE.

Recovery from the log proceeds as follows. The failed replica scans log entries serially, starting from the head of the log. While scanning, each PREPARE entry in the Log is inserted into a hash table, indexed by the entry's timestamp. Whenever a COMMIT entry with the timestamp is encountered, the corresponding PREPARE entry is removed from the hash table and executed. When the replica finishes scanning the log, it has executed all committed commands in timestamp order. However, towards the end of Log there might be some PREPARE entries which do not have the corresponding COMMIT in Log. To recover these entries, a replica sends RETRIEVECMDS to a majority of replicas in $Spec$, and only executes commands that have been logged by a majority. Finally, the replica triggers reconfiguration to join the current configuration, using Algorithm 3. Checkpointing can be used to avoid replaying the whole log and speed up the recovery process.

C. Discussion

The overall time to exclude a failed replica from the current configuration is the time to detect the failure, plus the time to reconfigure. Reconfiguration requires one initial exchange with a majority for SUSPENDing the replicated state machine, one exchange with a majority to agree on a timestamp and the set of commands that could have been committed. Some replicas might require an additional exchange for STATETRANSFER. When replicas are deployed at multiple data centers, the timeout for failure detection normally dominates the reconfiguration duration, similar to other existing protocols.

In practice failures within a replication group do not happen often, because the replication degree of a service is normally small, such as five or seven, and data centers have redundant network connections to the Internet. Therefore, we do not expect reconfiguration to be triggered frequently and affect the availability of the replicated service.

Temporary latency variations due to network congestions on the paths between data centers may affect the commit latency of Clock-RSM. A managed WAN among data centers, which provides stable network latency, can solve this problem [8].

VI. EVALUATION

We evaluate the latency of Clock-RSM and other protocols with experiments on Amazon EC2 and numerical analysis. Our evaluation shows that: 1) Clock-RSM provides lower latency than Mencius-bcast. 2) With five and seven replicas, Clock-RSM provides lower latency than Paxos-bcast at the non-leader replicas in most cases, and it provides

similar or slightly higher latency at the leader replicas. 3) With three replicas, a special case for Paxos-bcast, Clock-RSM provides similar or slightly higher (about 6% on average) latency than Paxos-bcast at all replicas.

We also evaluate the throughput of the four protocols on a local cluster. Our results show that: 1) Clock-RSM and Mencius have similar throughput for all command sizes. 2) They provide higher throughput than Paxos and Paxos-bcast for commands of large size while their throughput is lower for small and medium size commands.

A. Implementation

We implement Clock-RSM, Paxos, Paxos-bcast and Mencius-bcast in `C++`, using Google's Protocol Buffers library for message serialization. The implementation is event-driven and fully asynchronous. The protocol is divided into steps and each step is executed by a different thread. When a thread of one step finishes processing a command, it pushes the command to the input queue of the next step. The thread executing a step batches the same type of messages being processed whenever possible. However, to avoid increasing latency, it does not wait intentionally to batch more messages.

To evaluate the protocols, we also implement an in-memory key-value store which we replicate using the above protocols. In all experiments, clients send commands to replicas of the key-value store to update the value of a randomly selected key. Each protocol replicates the update commands and executes them in total order.

For Clock-RSM, we run NTP to keep the physical clock at each replica synchronized with a nearby public NTP server. With the assistance of `clock_gettime` system call in Linux, we obtain monotonically increasing timestamps.

B. Latency in wide area replication

We evaluate the latency of Clock-RSM and compare it with other protocols with three and five replicas. We place the replicas at Amazon EC2 data centers in California (CA), Virginia (VA) and Ireland (IR), plus Japan (JP) and Singapore (SG) for the five-replica experiments. To help reason about experiment results and numerical analysis later, we measure the round-trip latencies between data centers using `ping` and report them in Table III.

We run both replicas and clients on large EC2 instances that run Ubuntu 12.04. The typical RTT in an EC2 data center is about 0.6ms. There are 40 clients issuing requests of 64B to a replica at each data center. Clients send requests in a closed loop with a think time selected uniformly randomly between 0 and 80ms. We enable the extension of Clock-RSM given in Algorithm 2 and set Δ to 5ms. We consider two types of workloads: With balanced workloads all replicas serve client requests; with imbalanced workloads only one replica serves client requests.

	VA	IR	JP	SG	AU	BR
CA	83	170	125	171	187	212
VA	-	101	215	254	220	137
IR	-	-	280	216	305	216
JP	-	-	-	77	129	368
SG	-	-	-	-	188	369
AU	-	-	-	-	-	349

Table III Average round-trip latencies (ms) between EC2 data centers. CA, VA, EU, JP, SG, and BR correspond to California, Virginia, Ireland, Japan (Tokyo), Singapore, Australia, and Brazil (São Paulo), respectively.

1) **Balanced workloads:** The first group of experiments use balanced workloads, where clients of each replica simultaneously generate requests.

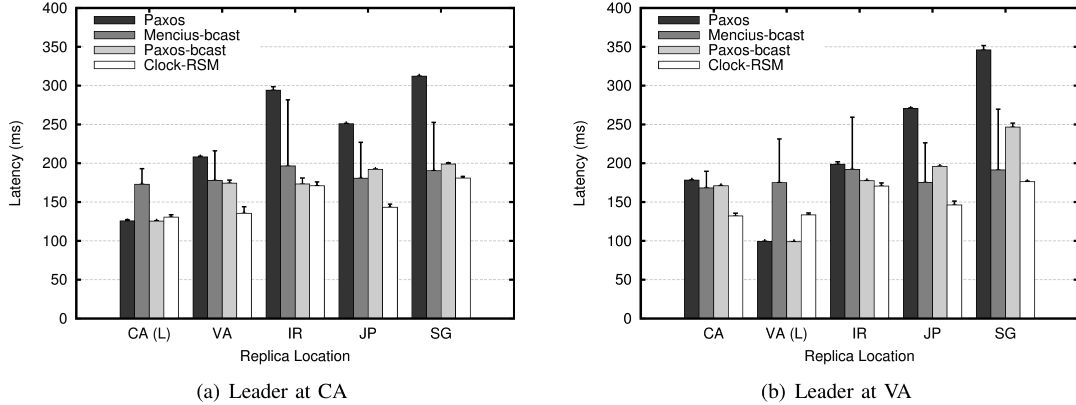
Figure 1 shows the average and 95%ile (percentile) latency at each of five replicas. Designating the replica at VA as the leader gives the best overall latency for Paxos and Paxos-bcast. Clock-RSM provides lower latency at all replicas except the leader of Paxos and Paxos-bcast.

For Paxos and Paxos-bcast, a leader replica needs only one round trip to a majority replicas to commit a command. For Clock-RSM, a replica requires contacting a majority plus the extra latency possibly introduced by the overlapping stable order and prefix replication processes. In these two experiments, the stable order process contributes to the commit latency because the highest latencies between two replicas are quite significant. The round-trip latency between JP and IR is up to 280ms. This means the command latency at JP and IR is at least 140ms. As a consequence, Clock-RSM has higher latency at leader replicas. However, the extra latency contributed by command ordering in Clock-RSM is smaller than the latency of forwarding a command from a non-leader replica to the leader in Paxos and Paxos-bcast. Hence Clock-RSM provides lower latency at all non-leader replicas.

We point out that the highest latency of Clock-RSM at all replicas is lower than Paxos and Paxos-bcast. The latencies of Clock-RSM at all replicas are more uniform. For the average latency of all replicas, Clock-RSM is also better.

Clock-RSM provides lower latency than Mencius-bcast at all replicas. The 95%ile latency of Mencius-bcast is much higher than its average, because the commit of a command may be delayed by another concurrent command from a different replica. The delay varies from zero up to one-way latency between two replicas. Paxos-bcast is also better than Mencius-bcast in most cases. Mencius-bcast sometimes provides lower latency than Paxos at non-leader replicas, because it requires fewer steps and concurrent commands from all replicas help the stable order process.

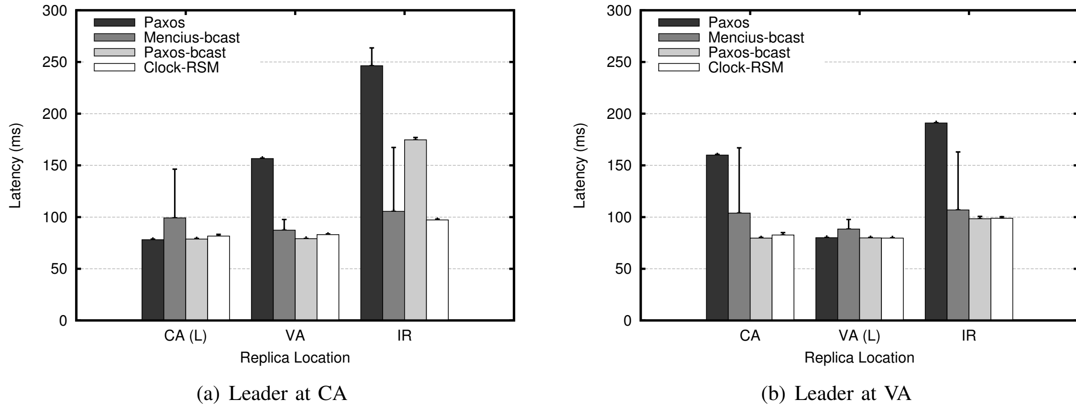
Figure 2 shows the average commit latency and 95%ile latency at each of three replicas. Designating the replica at VA as the leader gives the best overall latency for Paxos and Paxos-bcast.



(a) Leader at CA

(b) Leader at VA

Figure 1 Average (bars) and 95%ile (lines atop bars) commit latency at each of five replicas. Workload is balanced.



(a) Leader at CA

(b) Leader at VA

Figure 2 Average (bars) and 95%ile (lines atop bars) commit latency at each of three replicas. Workload is balanced.

A three-replica setup is a *special case* for both Clock-RSM and Paxos-bcast. For Paxos-bcast, the leader replica commits a command after it receives the logging confirmation from the nearest replica. For a non-leader replica, it first forwards the command to the leader and then waits for the logging confirmation from a majority. Most likely after it logs its own command, the logging confirmation of the leader arrives, if triangle inequality still holds with latencies. Hence all replicas in Paxos-bcast require one round trip to another replica to commit a command. When we designate the replica with the smallest weighted degree as the leader, Paxos-bcast always needs one round trip to the nearest replica to commit a command.

For Clock-RSM, prefix replication does not affect latency anymore because it is dominated by stable order. A replica commits a command after it receives the corresponding PREPAREOK from the nearest replica (majority replication) and a greater timestamp from the farthest replica (stable order). For the three locations in this experiment, the highest latency (between VA and IR) is roughly twice of the lowest (between CA and VA). Hence Clock-RSM also needs one round trip to the nearest replica to commit a command.

In Figure 2(a), the Paxos-bcast leader (CA) and its nearest non-leader replica (VA) have similar commit latency to Clock-RSM since they both require one round-trip messages to the nearest replica. For the other non-leader replica (IR), it has to use the longest path. Hence, the commit latency is much higher than Clock-RSM. In Figure 2(b), the Paxos-bcast leader (VA) avoids the longest path. Both protocols require one round trip to the nearest replica. Hence they have similar latencies at all replicas.

To further clarify the latency characteristics of each protocol, we also present their latency distributions. Figure 3 shows the latency distribution at JP when there are five replicas and the leader is at CA. Both Paxos and Paxos-bcast have very predictable latency as the commit of a command is not affected by other commands. The latency of Mencius-bcast varies from 134ms to 230ms because of the delayed commit problem. Clock-RSM has some variance because, with this particular layout, the latency required by prefix replication sometimes dominates. Figure 4 shows the latency distribution at CA when there are three replicas and the leader is at VA. The results are similar to the ones in Figure 3 except that, with this replica layout, the latency of Clock-RSM almost does not vary, because prefix replication is dominated by the stable order process.

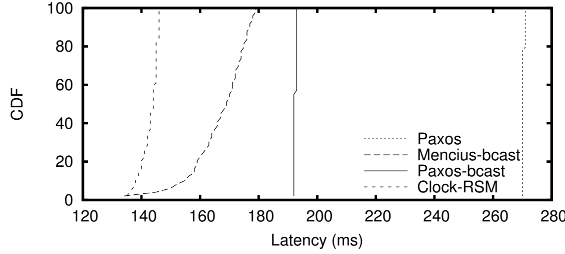


Figure 3 Latency distribution at JP with five replicas. The leader is at CA. Workload is balanced.

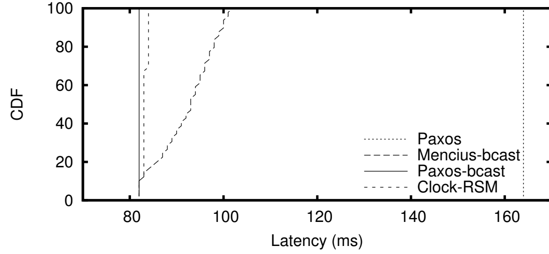


Figure 4 Latency distribution at CA with three replicas. The leader is at VA. Workload is balanced.

2) **Imbalanced workloads:** We next evaluate the latency of the four protocols under imbalanced workloads. For each run of the experiment, clients issue requests to only one replica. Figure 5 shows the results for five replicas. This experiment is the same as the one used in Figure 1(a) except that the workload is imbalanced.

Paxos and Paxos-bcast provide the same latency for both balanced and imbalanced workloads. Clock-RSM also provides similar predictable latency to both imbalanced and balanced workloads, because of the PREPAREOKs of previous commands and CLOCKTIMEs that carry latest clock time of other replicas. The average latency of Mencius-bcast becomes much higher when it has imbalanced workloads. This is because Mencius-bcast needs to receive logging acknowledgement with skipped rounds from every replica to make sure that other replicas do not propose a command in a previous round. The 95%ile latency is close to the average because the delayed commit problem does not happen when there is no concurrent command at any replica. Figure 6 shows the latency distribution.

In summary, with realistic latencies among data centers, Clock-RSM provides lower latency in most cases. The experiment results above also confirm our analysis in Section IV.

C. Numerical comparison of latency

Our above experiments on EC2 show that Clock-RSM provides lower latency than others in most cases with two groups of replicas of different sizes. To complete the evaluation, we compare Clock-RSM with Paxos-bcast numerically with all the possible data center combinations on EC2.

We use all the combinations of three, five, and seven replicas located at different EC2 data centers from Table III.

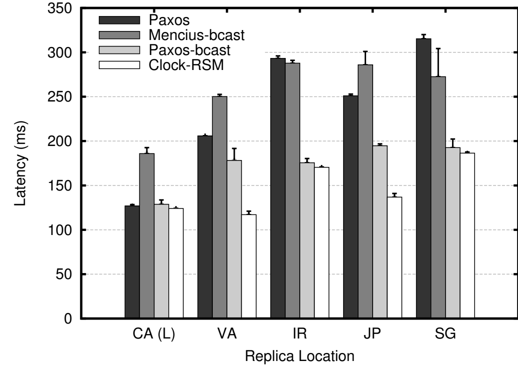


Figure 5 Average (bars) and 95%ile (lines atop bars) commit latency at each of five replicas. The leader of Paxos and Paxos-bcast is at CA. Workload is imbalanced.

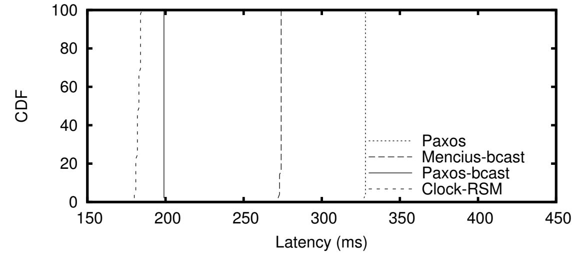


Figure 6 Latency distribution at SG with five replicas. The leader of Paxos and Paxos-bcast is at CA. Workload is imbalanced.

We plug the measured latencies in Table III into the latency formulas in Table II. Paxos-bcast always chooses the best leader replica that provides the lowest average latency of all replicas in the group.

Figure 7 shows the average latency of replicas from all groups of the same size. We compute two types of average latency: average *all* latency includes latencies at all replicas of a group while average *highest* latency only includes the latency at one replica that provides the highest latency in the group. As the figure shows, Clock-RSM provides lower latency for both five and seven replicas. Its improvement for the average highest latency is greater, because for Paxos-bcast, latencies at different replicas are more spread. The latency at a non-leader replica is much higher than the leader replica, as it needs one extra message to forward a command to the leader. In contrast, latencies in Clock-RSM are closer to each other because every replica requires the same number of steps to commit a command.

With three replicas, Paxos-bcast is slightly better than Clock-RSM, because we always choose the best leader for it, which leads to optimal commit latency at all replicas. This also validates our previous analysis of the protocols with three replicas.

Table IV shows the latency reduction of Clock-RSM over Paxos-bcast at all replicas for different replication groups. For instance, for all replicas in the groups with five replicas,

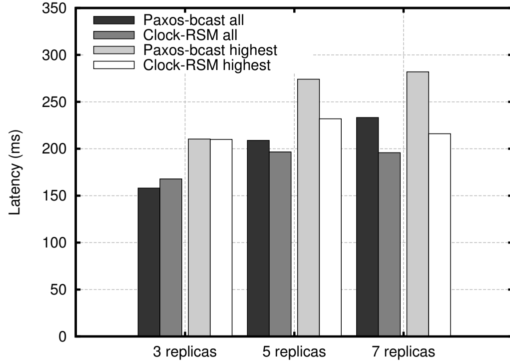


Figure 7 Average commit latency. *all* includes latencies at all replicas of a group while *highest* only includes the latency at one replica that provides the highest latency.

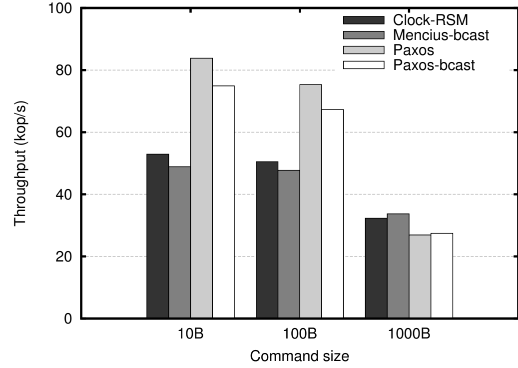


Figure 8 Throughput for small (10B), medium (100B), and large (1000B) commands with five replicas on a local cluster.

	Replica Percentage	Absolute Reduction	Relative Reduction
3 replicas	0.0%	0.0ms	0.0%
	100.0%	-9.9ms	-6.2%
5 replicas	68.6%	31.9ms	15.2%
	31.4%	-30.6ms	-14.6%
7 replicas	85.7%	50.2ms	21.5%
	14.3%	-39.4ms	-16.9%

Table IV Latency reduction of Clock-RSM over Paxos-bcast. Negative latency reduction means Clock-RSM provides higher latency.

the latency of Clock-RSM at 68.6% of the replicas is lower than Paxos-bcast. On average, it reduces the latency by 31.9ms, i.e., by 15.2%, at those replicas. For 31.4% of the replicas, the latency of Clock-RSM is higher. On average, it increases the latency by 30.6ms, i.e., by 14.3%, at those replicas. We look into all these 31.4% replicas and find that most of them are the leader replica in their group and a few are the non-leader replica that is very close to the leader. For groups with seven replicas, we have similar results. For groups with three replicas, the latency of Paxos-bcast is slightly better, because it provides optimal latency in this special case.

D. Throughput on a local cluster

Although the goal of Clock-RSM is to provide low commit latency in a WAN environment, for completeness, we also evaluate its throughput and compare it with other protocols. Our experimental results show that Clock-RSM has competitive throughput.

To avoid the network bandwidth limit on EC2 across data centers, we run experiments on a local cluster. Each server has two Intel Xeon processors with 4GB DDR2 memory. All servers are connected to a single Gigabit Ethernet switch. A replica runs on one server exclusively. Replicas log commands to main memory to avoid the disk being the bottleneck. Clients send frequent enough commands to all replicas to saturate them.

Figure 8 reports the throughputs for five replicas and command sizes of 10B, 100B, and 1000B. In all cases, CPU is the bottleneck and message sending and receiving is the major consumer of CPU cycles.

For 10B and 100B commands, Paxos and Paxos-bcast have higher throughput than Mencius-bcast and Clock-RSM. This is because all the non-leader replicas forward commands to the leader in Paxos and Paxos-bcast, which can batch more commands when sending and receiving messages. For 1000B large commands, Paxos and Paxos-bcast have lower throughput. The leader replica becomes the performance bottleneck, because batching large messages does not help throughput anymore.

Clock-RSM and Mencius have similar throughput as they have the same communication pattern and message complexity. Paxos provides better throughput than Paxos-bcast because its message complexity is lower. But the improvement is not significant since Paxos requires one more message step.

Our measurements do not support the claim that a multi-leader protocol always provides better throughput than Paxos because the leader in Paxos is the performance bottleneck [16, 17]. When replicas batch messages opportunistically, without waiting intentionally, the leader replica of Paxos has more chances to batch and hence increases throughput in the case of small and medium commands. Prior work evaluates throughput using different implementations or configurations. For the evaluations in Mencius [16], replicas do not batch messages. For the experiments in Egalitarian Paxos [17], although replicas batch messages, the Paxos leader handles all client messages while the other replicas only process replication messages.

VII. Related Work

In addition to Multi-Paxos [12, 13] and Mencius [16], we compare Clock-RSM with other existing work.

Fast Paxos [15] allows clients to send commands directly to all replicas to reduce commit latency. In good runs,

it requires two message delays to commit a command. However, under collisions due to concurrent proposals, Fast Paxos requires at least two additional messages for collision recovery. Collisions are frequent in a geo-replicated environment with balanced workloads, and thus Fast Paxos results in significantly higher latency than Clock-RSM.

Some protocols relax the total order property of state machine replication. Generalized Paxos [14] and Generic Broadcast [18] commits commands that do not interfere out of order in one round trip. It requires a stable leader to order interfering commands, which takes at least two additional round trips. Egalitarian Paxos [17], also called EPaxos, does not require a designated leader. Every replica in EPaxos can serve client requests and submit commands. EPaxos also commits non-interfering commands out of order in two message delays, which is one round-trip latency to (at least) a majority of replicas. The slow path, which resolves conflicts, requires one additional round trip. EPaxos provides linearizability. However, local reads in EPaxos may see updates in different orders at different replicas. In contrast, Clock-RSM provides linearizability and maintains an explicit total order over updates. Database replication, one of the most popular applications of state machine replication, often requires total order replication to maintain strong transaction isolation levels, such as serializability [4] and snapshot isolation [3], as in a single-copy database [9].

MDCC [10] uses Generalized Paxos to build a replicated partitioned key-value store across data centers. MDCC reduces replication latency by running one instance of Generalized Paxos per key. An update to a key commits in one round trip using the fast path, under the assumption that conflicting updates on the same key are rare. However, MDCC provides "read committed" guarantee to transactions, a weaker isolation level than both serializability and snapshot isolation. In contrast, Clock-RSM can be used for total order replication across all keys while providing low latency and strong transaction isolation.

Using physical clocks for state machine replication is discussed in [11] and [19]. However, neither of them provides a complete solution. Clock-RSM is a clearly specified protocol with reconfiguration for failure handling. It relies on physical clocks to reduce replication latency across data centers.

Spanner [6] uses physical clocks to provide linearizable transactions across replicated partitions. It relies on Multi-Paxos to replicate each partition. Replica leaders order transactions with physical timestamps. Clock-RSM is a new state machine replication protocol and uses physical clocks to improve latency. The correctness of Spanner depends on synchronized clocks with bounded skew while Clock-RSM requires clocks to be only loosely synchronized.

An atomic broadcast algorithm using physical clocks is introduced in [20]. This algorithm relies on ordinary broadcast and generic broadcast and delivers a message with two message delays in good runs. Similar to Clock-RSM, each replica coordinates its own commands and commands are totally ordered by physical time intervals. In contrast, Clock-RSM is a simpler and more efficient state machine replication protocol that includes recovery and reconfiguration and targets realistic geo-replication environments.

VIII. Conclusion

We introduce Clock-RSM, a state machine replication protocol that pushes the latency limit of strongly consistent replication. Clock-RSM relies on loosely synchronized clocks to reduce latency. We evaluate our protocol extensively with realistic workloads, where latencies among data centers are non-uniform. We show that, compared with state of the art protocols, Clock-RSM reduces latency in most cases with real world replica placements.

References

[1] The network time protocol. http://www.ntp.org, 2014.

[2] J Baker, C Bond, JC Corbett, et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

[3] H. Berenson, P. Bernstein, J. Gray, et al. A critique of ansi sql isolation levels. In *SIGMOD*, 1995.

[4] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. 1986.

[5] Brad Calder, Ju Wang, Aaron Ogus, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[6] James C Corbett, Jeffrey Dean, Michael Epstein, et al. Spanner: Googles globally-distributed database. In *OSDI*, 2012.

[7] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 1990.

[8] Sushant Jain, Alok Kumar, Subhasree Mandal, et al. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[9] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB*, 2000.

[10] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *EuroSys*, 2013.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 1978.

[12] Leslie Lamport. The part-time parliament. *TOCS*, 1998.

[13] Leslie Lamport. Paxos made simple. 2001.

[14] Leslie Lamport. Generalized consensus and paxos. 2004.

[15] Leslie Lamport. Fast paxos. In *Distributed Computing*. 2006.

[16] Yanhua Mao and Flavio P Junqueira. Mencius: Building efficient replicated state machines for wans. In *OSDI*, 2008.

[17] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *SOSP*, 2013.

[18] Fernando Pedone and André Schiper. Generic broadcast. In *Distributed Computing*. 1999.

[19] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 1990.

[20] Piotr Zieliński. Low-latency atomic broadcast in the presence of contention. 2008.

APPENDIX

In this section we provide proof sketches for Algorithms 1 and 3. We first argue that replicas execute commands in the same order and that every correct replica executes every command (agreement). Finally, we show that linearizability, as defined in Section II-B, follows from the above properties.

Claim 1. *If any replica r_m executes command c_i followed by command c_j with timestamps ts_i and ts_j respectively, then $ts_i < ts_j$.*

Proof sketch. Suppose replica r_m executes command c_i (at line 16 of alg. 1). We distinguish two cases:

Case (a): r_m's $PendingCmds$ contains $\langle c_j, ts_j, k\rangle$. Notice that function COMMITTED(ts) evaluates to *true* only if the given timestamp ts is the smallest timestamp among all commands in $PendingCmds$ (line 23, alg. 1). Since c_i is executed, the invocation COMMITTED(ts_i) must have returned *true*. Therefore $ts_i < ts_j$.

Case (b): r_m's $PendingCmds$ does not contain $\langle c_j, ts_j, k\rangle$. Assume c_j originated at replica r_k. By line 22 of alg. 1, $ts_i \leq LatestTV[k]$, which implies r_m received a PREPARE or a PREPAREOK message from r_k tagged with a timestamp greater than ts_i. Since channels are FIFO and messages are sent in timestamp order, $LatestTV[k] < ts_j$. Therefore $ts_i < ts_j$.

Claim 2. (Total order). *If a replica executes commands c_i and c_j, in this order, then no replica executes c_j before c_i.*

Proof sketch. By Claim 1, any replica executes commands in clock timestamp order. What remains to show is that the timestamp order is a total order. Replicas assign monotonically increasing clock timestamps to each command. The clock timestamp, together with the unique replica *id* forms a total order. Moreover, timestamps are assigned once by one replica, and never change.

Claim 3. *If any replica executes command c in epoch e, then every correct replica in epoch $e + 1$ has executed c.*

Proof sketch. Assume that some replica has executed command c with timestamp ts in epoch e. It means that a majority of replicas has acknowledged a message $\langle$PREPARE $c, ts\rangle$ and logged it to their stable storage.

Let r be the replica that triggers function RECONFIGURE (Algorithm 3) in epoch e and whose proposal (line 6, alg. 3) is decided by all correct replicas. We next consider cts, the largest commit timestamp in r's Log, and distinguish two cases:

Case (a): $ts > cts$. That is, r has not executed c yet. In this case r fetches all commands with timestamps greater than cts from a majority of replicas (lines 4-5, alg. 3). Since any two majorities intersect it must be that at least one replica returned command c to replica r. Replica r included command c in its consensus proposal (line 6, alg. 3), and all correct replicas eventually deliver c.

Case (b): $ts \leq cts$. In this case r has already executed c and therefore does not include c in its proposal. All correct replicas eventually deliver the consensus decision for epoch $e + 1$. If a replica has not executed c yet, its last commit mark in the log is smaller than ts (line 13, alg. 3), in which case it fetches c from a majority of replicas in function STATETRANSFER.

In both cases, a replica has either already executed c or its set of commands $cmds$ after line 14 of alg. 3 includes c, in which case c is executed in lines 16-20 of alg. 3 before transitioning to epoch $e + 1$.

Claim 4. (Agreement) *If a replica executes command c, then every correct replica eventually executes c.*

Proof sketch. Suppose replica r executes command c. We have to show that every correct replica eventually executes c, both during normal case operation and across subsequent epochs. Algorithm 1 ensures that during normal case operation every replica eventually delivers PREPARE message and therefore replicas include c in their set of pending commands. Since timestamps are monotonically increasing, it must be that c eventually becomes the command with smallest timestamp (line 23, alg. 1), and that all replicas have proposed or reported higher timestamps (line 22, alg. 1). Finally, every replica will receive enough acknowledgments (line 21, alg. 1) to execute c. In case of failures, we rely on a correct replica that triggers function RECONFIGURE. And by Claim 3, any command that has committed in the current epoch will be executed by every correct replica before transitioning to the subsequent epoch.

Claim 5. (Linearizability) *Clock-RSM is linearizable.*

Proof sketch. Let σ be an execution of client commands that consists of $\langle$REQUEST $cmd\rangle$ and their corresponding

$\langle$REPLY $result\rangle$. We have to show that there exists a permutation π of σ such that: 1) π respects the semantics of the commands, as defined in their sequential specification; and 2) π respects the real-time ordering of commands across all clients. Let π be a permutation of σ ordered according to the clock timestamp ordering provided by Algorithm 1.

We first show that π respects the sequential semantics of the commands. The replication protocol executes commands in total order (Claim 2), and replicas execute each and every command (Claim 4). Which means that every replica executes the *upon* clause of lines 14-19 of alg. 1 for each command in the same order. Moreover, every command is execute serially, one command at a time, thus satisfying the semantics of commands.

We next claim that π satisfies the real-time ordering of commands in σ. Suppose command c_i finishes before command c_j begins in σ. This implies that c_i's client has received a reply for c_i before c_j is submitted to a replica. Obviously, Algorithm 1 orders c_j after c_i. Thus c_i precedes c_j in π.