2017 IEEE 37th International Conference on Distributed Computing Systems

MeteorShower: Minimizing Request Latency for Majority Quorum-based Data Consistency Algorithms in Multiple Data Centers

Ying Liu, Xi Guan, Vladimir Vlassov, and Seif Haridi
Department of Software and Computer Systems
KTH Royal Institute of Technology, Sweden
Email: yinliu@kth.se, xig@kth.se, vladv@kth.se, haridi@kth.se

***Abstract*—With the increasing popularity of serving and storing data in multiple data centers, we investigate the efficiency of majority quorum-based data consistency algorithms under this scenario. Because of the failure-prone nature of distributed storage systems, majority quorum-based data consistency algorithms become one of the most widely adopted approaches. In this paper, we propose the MeteorShower framework, which provides fault-tolerant read/write key-value storage service across multiple data centers with sequential consistency guarantees. A major feature is that most read operations are executed locally within a single data center. This results in lowering read latency from hundreds of milliseconds to tens of milliseconds. The data consistency algorithm in MeteorShower augments majority quorum-based algorithms. Thus, it keeps all the desirable properties of majority quorums, such as fault tolerance, balanced load, etc.**

An implementation of MeteorShower on top of Cassandra is deployed and evaluated in multiple data centers using the Google Cloud Platform. Evaluations of MeteorShower framework have shown that it can consistently serve read requests without paying the communication delays among replicas maintained in multiple data centers. As a result, we are able to improve the latency of read requests from hundreds of milliseconds to tens of milliseconds while achieving the same latency on write requests and the same fault tolerance guarantee. Thus, MeteorShower is optimized for read intensive workloads.

***Keywords*—Geo-distributed storage systems; Majority quorum; Data consistency; Synchronized clocks; Distributed time**

I. Introduction

With the growing popularity of Internet-based services, more powerful backend storage systems are needed to match ever increasing workloads in terms of concurrency, intensity, and locality. Under this context, distributed storage systems with global data replication are proposed. On the one hand, the flexibility to increase or decrease the number of data replicas and replication servers allows the storage systems to handle workloads with different intensities. On the other hand, the possibility to allocate data replicas in different data centers enables the storage systems to serve requests according to their initiated locality, which significantly reduces the service latency. However, data replication comes with the overhead of maintaining data consistency.

When data replication is applied in a data store, whether data replicas are consistent with each other at any given time is defined according to the data consistency model. Replicas diverge when they are accessed and modified by various clients in a concurrent fashion. It is usually laborious and also not necessary to guarantee that replicas are consistent strictly at any time. Based on different requirements from the upper applications that interact with the data store, different data consistency models are introduced.

The data consistency model is a contract between a (distributed) data store and processes, in which the data store specifies precisely what the result of read and write operations are in the presence of concurrency. Simply put, it describes under which circumstances, the replicas are consistent, and when they are not. Representative data consistency models include linearizability, sequential consistency, causal consistency, FIFO consistency, etc. In this work, we focus on systems that adopt sequential consistency model, which is one of the most widely used data consistency models.

More specifically, we focus on a classic approach to achieve sequential data consistency model in a failure-prone environment, i.e. the majority quorum. A replica quorum consists of all the replicas of a data item. The size of a replica quorum equals the replication degree (n). Let us assume that a read request on a data item X is served by r number of replicas and a write request on X is served by w number of replicas, then the minimum requirement to achieve sequential data consistency in a majority quorum approach is $r + w > n$. Typically, a read/write request is sent to all replicas in order to obtain a sufficient amount of replies (r and w) to satisfy the requirement. This approach yields adequate performance when replicas are relatively close to each other, e.g., in the same data center. However, this is not the case in a geo-replicated storage system, where replicas are hosted in different data centers for performance and availability purposes. Communications among highly distributed replicas incur significant delays. Consequently, using majority quorums to achieve sequential data consistency when replicas are deployed geographically leads to significant increase in request latency.

A. Problem Formation

In this section, we investigate the cause and provide insights on high request latency while using majority quorums in a geo-distributed data store.

The setup: We assume a replicated data store deployed in multiple data centers and replicas are not hosted in the same data center. A data center hosts many storage instances, which are responsible for hosting a specific part of the total

1063-6927/17 $31.00 © 2017 IEEE
DOI 10.1109/ICDCS.2017.112

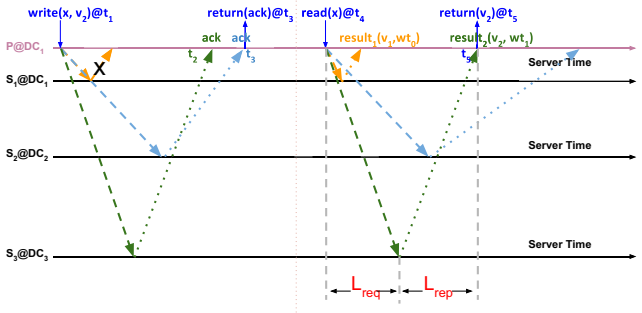

Fig. 1: Read and write quorum operations showing a proxy at DC_1 and three storage instances at DC_1, DC_2 and DC_3

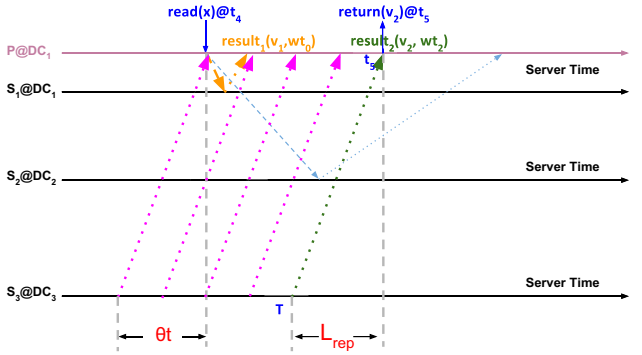

Fig. 2: Read operation showing periodic status messages

namespace. A storage instance manages various data items, which are replicated among different storage instances hosted in different data centers. The data store provides a key-value query interface to clients. Client requests are received and returned by storage instances from the closest data center.

A scenario: Figure 1 illustrates a scenario where the requested data item is replicated in three data centers, i.e. DC_1, DC_2 and DC_3. A client close to DC_1 has first issued a write request, which has been received by one of the storage instances in DC_1, which then acts as a proxy for the request. The request writes a value v_2 of x with its request timestamp t_1 on replicas when the value of x stored locally is older than t_1. The write returns when a majority of replicas acknowledge the write. A read is returned when it has received the replies from a majority of the replicas. In our case, DC_1 has returned a value v_1 with timestamp t_1 while DC_3 has returned a value v_2 with timestamp t_2. Assuming $t_2 > t_1$, then v_2 is returned by the read. We use this representative scenario to explain the causes and insights of high request latency.

The cause: The essential cause for high read request latency is the network delay on requesting item values from all replicas (e.g., L_{req} in Figure 1) and replying the requests (shown as L_{rep} in Figure 1). Essentially, L_{req} is paid to ask all the replicas, involved in a specific request, to report their current status. L_{rep} is paid to deliver the status of all replicas to the requester, i.e. the proxy.

The insight: We can avoid remote reads and only perform local reads if the replicas send periodically their current status. Figure 2 continues the above scenario with the focus on the read request. The remote request message from the proxy to DC_3 is removed while periodic status messages from DC_3 to the proxy are added. Given perfectly synchronized clocks, if the proxy waits for the status message sent from DC_3 at T, the reply of the read request would be the same as the previous scenario. We are going to show that we can preserve sequential consistency while serving the read with data on DC_3 earlier than T. Specifically, we are going to show that the read can be served with data on DC_3 up to θt old with respect to its request timestamp t_4. The maximum value of θt, which defines the maximum staleness of valid status messages, is derived in later sections. Using the maximum value of θt dramatically reduces the read latency when replicas are deployed in multiple data centers.

The contributions of the paper are the following:

- We identify the cause for high latency requests when implementing stronger data consistency models, such as sequential consistency, in a geo-distributed data store;
- We propose a novel fault-tolerant algorithm that performs local reads and preserves sequential consistency, which minimizes the read request latency in geo-distributed data stores;
- We prove the correctness of the algorithm;
- We have implemented and evaluated our algorithm as a framework, called MeteorShower, in a multiple data center setup. We have tested MeteorShower on various workloads that show realistic reductions in latency on the workloads.

II. Algorithm Design

Before describing our algorithm that minimizes quorum reads in geo-distributed storage systems, we present our assumptions of the system model and the definition of sequential consistency, which is achieved by our algorithm.

A. System Model

We assume a system consisting of a set of storage instances connected via a communication network. Messages in this communication network can be delayed or lost, but cannot be corrupted. Each storage instance replicates a portion of the total data in the system. There is a storage process running at each storage instance. The storage process has the whole namespace mapping and addresses of replicas. The process is able to access the data stored locally.

Furthermore, each process has the access to its local physical clock. Formally, a clock is a monotonically increasing function of real time [1]. We assume that the time differences caused by clock drifts among different servers are bounded to an error ε, which can be achieved by using the Network Time Protocol (NTP). It means that the clock-time differences between any two instances, $C_1(t)$ and $C_2(t)$, is bounded to ε, i.e., $|C_1(t) - C_2(t)| \leq \varepsilon$.

B. Sequential Consistency

For the formal exposition, we define an execution to be sequences of request and response events. **A sequential execution** is an execution where each request is followed by the corresponding response. And, **a legal execution** is a sequential one where each read returns the last write value. Then, we provide the formal definition of sequential consistency (SC) based on the definition given by Hagit Attiya et. al [2].

Definition 1. *An execution δ is sequentially consistent if there exists a legal execution γ such that γ is a permutation of δ and, for each process p, $\delta|p$ is equal to $\gamma|p$.*

SC preserves the real-time order of operations by the same process [3]. Given the definition, we will design an algorithm that orders all writes and inserts reads to satisfy the legality condition while preserving process order.

First, in Section II-C, we describe our algorithm for achieving multiple data center sequential consistency, which performs local reads and majority writes. For simplicity, we explain our algorithm under the deployment that the storage namespace is partitioned identically in each data center. There are well-known namespace mapping techniques to extend it to handle a more general case, which is not the focus of our algorithm. Also, we assume perfectly synchronized time among replicas at this moment. Then, in Section II-D, we prove the SC guarantee of our algorithm while solving the constraint for status messages in order to be used for local reads, as mentioned in Figure 2 and defined in Section I-A. Afterward, we generalize the constraint of status messages when other write APIs are employed and discuss the scenarios when time among replicas are not perfectly synchronized in Section II-E.

C. Algorithm Description

We explain the algorithm in three parts, i.e. handling writes, handling status, and handling reads. The main data structures used in our algorithm is listed in table I.

A write request is first received by a proxy, which acts as the coordinator of the request. For example, in Cassandra, the storage server that receives the request acts as the proxy server. Then, a request timestamp ts is generated for the write using the local current time of the proxy. Afterward, the proxy broadcasts the write to all the replicas of the concerned key as illustrated in Figure 3 (a). The write request is returned when the proxy has received acknowledgements from a majority of replicas.

When a local storage receives the write from the proxy, it executes the request following the algorithm described in Figure 3 (b). Specifically, it checks whether the wts in the request is larger than the $timestamp$ of the concerned data item stored locally. If so, an update on the data item is performed locally and the metadata of the update is added to a Status list, which is used to communicate updates among replicas periodically. Then, an acknowledgement is sent back to the proxy. In this way, all writes are ordered deterministically based on their request timestamps, breaking ties with node IDs.

Replicas communicate their updates, only the metadata, using status messages as illustrated in the algorithm in Figure 4. Essentially, the algorithm has a sender and a receiver function.

Data: $write(key, value)$
Result: Majority write from the proxy
$ts = self.currentTime()$
foreach $r \in replicaList$ **do**
  Send $write(key, value, ts)$ to r
end
$ackNum = 0$
Receive ack with $timeout$ do
if $timeout$ **then**
  Return $timeout$
end
if ack **then**
  $ackNum = ackNum + 1$
  if $ackNum > \frac{replicaList.size}{2}$ **then**
    Return $Success$
  end
end
/* A majority write to all replicas
   to ensure data availability. */

(a) Handle Write Requests - Proxy Side

Data: $writeLocal(key, value, wts)$
Result: Write to the local store and ack the proxy
Receive $write(key, value, wts)$ **from** o do
if $wts > localStore(key).timestamp$ **then**
  $localStore(key).write(value, wts)$
  $Status.append((key, wts))$
end
if $wts = localStore(key).timestamp$ ***and***
 $o.id > self.id$ **then**
  $localStore(key).write(value, wts)$
  $Status.append((key, wts))$
end
Send ack to o

(b) Handle Write Requests - Server Side

Fig. 3: Write Protocol

The sender function periodically broadcasts the metadata of the local updates, which are maintained in a list ($Status$), to remote replicas. Each entry (key, wts) in the $Status$ list is generated by a local update during the current dispatch interval. A status message associates the list of entries ($Status$) with a dispatch timestamp of the current interval. At the end of the interval, the status message is sent to remote replicas.

The receiver function receives status messages from remote replicas and merges them in a component called $statusMap$. $statusMap$ is a list of $replicaStatus$, which is a list of $((key, wts), sts)$. It means that the newest version of a key is updated at wts observed until sts at a certain replica. Thus, in some sense, $statusMap$ is able to provide a slightly outdated snapshot of remote replicas. Using the timestamps in the $statusMap$, our algorithm is able to decide whether a local version is updated enough to serve a read request, which is explained in the following paragraphs. Status messages also wake up pending read requests maintained in the $readSubscriptionMap$, which records a list of blocked reads because of lacking up-to-date status messages or data.

TABLE I: Data Structure used in the Algorithms

Data Structures	Variables and Functions	Description
Status	a list of (key, wts)	used to accumulate local updates in the interval between two consecutive status messages
LocalStore	an object to read and write key's $value$ and ts locally	used to access the locally stored data
StatusMap	a map from $replica$ to $replicaStatus$, where $replicaStatus$ is a list of $((key, wts), sts)$	used to record the status of remote replicas
ReadSubscriptionMap	a list of (key, ts)	used to keep reads that cannot be served currently
ReplicaList	a list of $replicaAddress$	used to maintain the addresses of replicas in the same replication group

Data: $broadcastStatus()$
Result: Broadcast updates to peer replicas periodically
foreach $dispatchInterval$ **do**
  $sts = self.currentTime()$
  foreach $r \in replicaList$ **do**
    Send $(Status, sts)$ to r
  end
  $Status.clear()$
end
/* The status message $(Status, sts)$ records the metadata of the local updates during the current $dispatchInterval$. */

(a) Broadcast Local Status

Data: $updateStatus(replica, (Status, sts))$
Result: Maintain status maps of remote replicas
Receive $(Status, sts)$ do
$statusMap(replica).update(Status, sts)$
foreach $(key, \_) \in Status$ **do**
  Wake reads on key in the $readSubscriptionMap$
end
/* $statusMap$ is used to keep track of updates on remote replicas. */

(b) Update Local Status

Fig. 4: Status Messages

We have to mention that status messages only contain the metadata of each update instead of the real data. This ensures the lightweight of the status messages and increases the flexibility of our algorithm since most of the replicated storage systems have their own data propagation mechanisms for replicas. We will continue our discussions on this design choice in the implementation section.

When a proxy receives a read, it forwards the read to the closest replica of the requested data item as shown in Figure 5 (a). A local read request is processed as illustrated by the algorithm in Figure 5 (b). Specifically, a local read does not initiate communications among remote replicas, which significantly minimizes the request latency. Instead, it checks the updates of remote replicas by analyzing the each replica's status recorded in the $statusMap$. A replica r's status is fetched by calling $statusMap(r)$, which is a list $(((key, wts), sts))$ recording the newest version (wts) of each key observed until sts on replica r. A read with request timestamp ts can be served locally when satisfying the following two constraints.

Data: $read(key)$
Result: Return the $value$ of key to the client
$ts = self.currentTime()$
Send $read(key, ts)$ to the closest replica r
Receive $result$ with $timeout$ do
if $timeout$ **then**
  Return $timeout$
end
if $result$ **then**
  Return $result$
end

(a) Handle Read Requests - Proxy Side

Data: $readLocal(key, ts)$
Result: Return the $value$ of key to the proxy
Receive $read(key, ts)$ **from** o do
$vs = [localStore(key).timestamp]$
foreach $r \in replicaList$ **do**
  Find e in $statusMap(r)$ such that $e = ((key, wts), sts)$
  if $sts + \theta t > ts$ **then**
    $vs.append(wts)$
  end
  /* θt is the maximum staleness of status messages in the consistency model */
end
if $vs.size > \frac{replicaList.size}{2}$ **then**
  $ts_{local} = localStore(key).timestamp$
  let $L = \{wts | wts \in vs \wedge ts_{local} \geq wts\}$
  if $L.size > \frac{replicaList.size}{2}$ **then**
    Send $localStore(key).value$ to o
  end
end
$readSubscriptionMap.append((key, ts))$
/* The read needs more remote status or updated local data to serve */

(b) Handle Read Requests - Server Side

Fig. 5: Read Protocol

First, we need a majority of valid statuses (vs). The local status is always a valid status. A status of a replica recorded in $statusMap$ has its sts larger than the read request timestamp ts subtract a θt is also a valid status. The θt is a variable that defines the staleness of statuses that can be used to serve a read request. The upper bound of θt defines the maximum tolerance of replica staleness while preserving sequential consistency. It is obvious that the choice of the upper bound θt provides the best performance of our algorithm. We derive the upper bound of θt while proving the sequential consistency of the algorithm in the next section.

Second, within the replicas having valid statuses, we compare their update timestamps wts regarding the requested key in the read. The $timestamp$ of the local copy needs to be greater or equal than at least half of the wts of the remote replicas. This means that there exists a majority quorum where the local replica has the value with the largest wts. Then, the local value is returned for the read. Otherwise, the read is kept in the $readSubscriptionMap$, which means that this read cannot be served locally at the moment because of lacking status messages or updated data. $readSubscriptionMap$ is iterated when status messages or data propagation messages are received.

D. Proofs

From the algorithm we provided, we prove that it satisfies sequential consistency under a specific constraint of θt.

Lemma 1. *For each execution δ and every process p, p's read operations reflect all the values successfully applied by its previous write operations, all updates occur in the same order at each process, and this order preserves the order of write operations on a per-process basis.*

Proof: We first prove that Lemma 1 holds when using majority reads and writes without the use of status messages as explained in Section I-A. Regarding a process p, a write returns only when a majority of replicas acknowledge the write proposed by p. This indicates that there is a majority of replicas that store a value at least as new as the value proposed by p. Since any read is served by querying a majority of replicas, a read by process p after the write will at least have one replica reflecting the value written by p or a newer value proposed by other processes. This is because a majority quorum intersects reads and writes and a read returns the value with the maximum timestamp. Since writes are applied on replicas based on their request timestamp, this guarantees a total order of all writes. Also, the order of writes that occur in the same process are preserved.

The write operation in our proposed algorithm, which uses status messages, is the same as the majority quorum write, and therefore the writes are ordered globally and per-process. The read operations are different. Now, we extend the above proof to the scenario of using status messages instead of querying a majority of replicas to serve reads. We derive the upper bound of θt so that we also satisfy the condition on reads stated in Lemma 1. We provide a scenario in Figure 6, where a read operation R of key X follows a write operation W of key X on the same process p with a delay of Δd. We assume that Δd is close to zero. The read operation R at least observes the

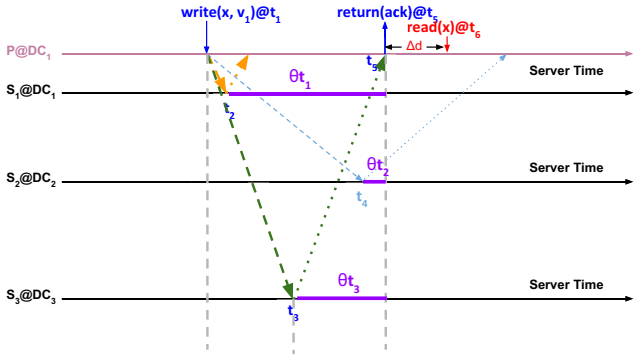

Fig. 6: Correctness condition of reads

previous write operation when it fetches the value of X after t_2 on DC_1, t_4 on DC_2 and t_3 on DC_3. We define θt_1, θt_2 and θt_3 to be the differences between the return time of W and the time when the write takes effect on the three data centers, i.e., t_2, t_4 and t_3. Thus, to satisfy the read condition in Lemma 1, values returned by R should be in the intervals of θt_1, θt_2 and θt_3 or higher on DC_1, DC_2 and DC_3 respectively. Since the read returns the value with the highest timestamp from a majority of the replies, we need to ensure that at least one value is valid in the replies. Assuming $\theta t_1 \geq \theta t_3 \geq \theta t_2$, then the staleness bound should be θt_3, which guarantees that reading from any majority quorum will include a valid value.

Mapping the staleness bounds to status messages, it means that status messages are valid for the read when the status message timestamps sts are larger than $t_5 - \theta t_3$. Obviously, the higher the value of θt_3 the better performance for reads we can achieve using status messages. Thus, we would like to explore the maximum value of θt_3 while preserving the read condition in Lemma 1. The read condition is preserved when the value fetched within θt_3 bound is always valid under any circumstances. Assume that the minimum message transmission delay between DC_3 and DC_1 is D. Since θt_3 is the period between the time when the previous write is applied on DC_3 and the time when the write is returned from $P@DC_1$, then D is the upper bound on the value of θt_3. Thus, given that θt_3 is bounded by D, our algorithm is able to satisfy the read condition in Lemma 1. ■

Lemma 2. *For each execution δ and every process p, p's read operations cannot reflect the updates applied by its following write operations.*

Proof: We provide a scenario in Figure 7, where a read operation R of key X precedes a write operation W of key X on the same process p with a delay of Δd. We assume that Δd is close to zero. We must show that the value returned by the read is associated with a write timestamp wt_2 less than the request timestamp t_6 of the write. From the figure, we know that $wt_2 \leq t_3$, so we have to show that $t_3 \leq t_6$. We have shown previously that θt_3 is greater than or equal to the minimum message transmission delay D between DC_3 and DC_1. Given Δd is close to zero, $wt_2 \leq t_3 \leq (t_5 - \theta t3) \leq (t_6 - \theta t3)$, which gives $t_6 \geq (wt_2 + \theta t3)$. Thus, the read returns a value before

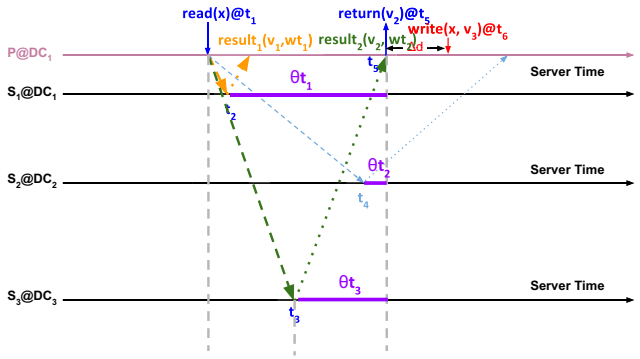

Fig. 7: Correctness condition of writes

the write, i.e., cannot observe the updates from the write. ■

Theorem 1. *The algorithm proposed provides sequentially consistent reads and writes from multiple processes.*

Proof: We follow the sequential consistency proof procedures provided in [2]. Fix an execution δ. Define the sequence of operations η as follows. Order the writes in η according to timestamps associated with each write requests in δ, breaking ties using process ids. Then, we explain the places to insert reads. We start from the beginning of δ. Read operations $[Read_p(X), Rep_p(X, v)]$ are inserted immediately after the latest of (1) the previous operation for p (either read or write on any object), and (2) the write that generated the value read by the read operation of p. Given a sequential execution σ then $ops(\sigma)$ is the sequence of operations in σ.

We must show $ops(\delta|p) = \eta|p$. We fix some process p and show $ops(\delta|p) = \eta|p$ in the following four scenarios.

1. The relative ordering of two reads in $ops(\delta|p)$ is the same in $\eta|p$ by the construction rules of η.

2. The relative ordering of two writes in $ops(\delta|p)$ is the same in $\eta|p$ since writes are ordered with request timestamps globally.

3. Suppose a relative ordering of read R follows write W in $ops(\delta|p)$. By construction of η and Lemma 1, R reads the value written by W.

4. Suppose a relative ordering of read R precedes write W in $ops(\delta|p)$. According to Lemma 2, the read R cannot read the value written by the write W.

Thus, we are able to prove that $ops(\delta|p) = \eta|p$, i.e., our algorithm provides sequential consistency. ■

E. Discussion of θt

We define the lower bound of θt to be 0 as all reads compare status message with their request timestamp. The upper bound of θt depends on the latencies between the proxy and storage servers as well as the write mode used, which can be "WRITE ONE", "WRITE QUORUM", or "WRITE ALL". We assume that the read and write modes are used correspondingly to achieve sequential consistency, i.e., $r+w > n$ as introduced in Section I. As we have derived above, when writes need a majority of replicas to acknowledge, the upper bound of θt is the median value of the latency lower bound between the proxy and storage servers. To generalize, when writes require only one replica to acknowledge, the upper bound of θt is the minimum latency lower bound between the proxy and storage servers. On the other hand, when writes require all replicas to respond, the upper bound of θt is the maximum latency lower bound between the proxy and storage servers. The application of the upper bound of θt provides the best performance of read requests.

In practice, the clock times among servers are not perfectly synchronized. Thus, the calculation of θt should also consider the worst-case clock drift among servers. Under our assumptions that the clock difference of any two servers' clock times $C_1(t)$ and $C_2(t)$ is bounded to ε, i.e., $|C_1(t) - C_2(t)| \leq \varepsilon$, we need to subtract ε on the calculation of θt in all above three cases.

Recall that the correctness condition for Lemma 2 is that $t_6 \geq (wt_2 + \theta t3)$. Taking the clock drift into account, we have $t_6 \geq (wt_2 + \theta t3 - \varepsilon)$. So, we have to ensure that $\theta t3 - \varepsilon \geq 0$ in order to have $t_6 \geq wt_2$, which is the correctness condition for Lemma 2.

To sum up, the correctness condition for our algorithm is $\theta t \geq \varepsilon$. In practice, the message transmission delays among data centers are around hundreds of milliseconds while the clock drifts among servers are around milliseconds. Thus, our algorithm is designed for replicated storage service across data centers.

III. Design of MeteorShower

A. Messages in MeteorShower

Using the algorithm above, we propose **MeteorShower**. It improves read latency for majority quorum based storage systems when they are deployed geographically. The major insight is that instead of pulling the status of remote replicas, MeteorShower enables replicas to report their status periodically through status messages. Then, MeteorShower judiciously uses the updates in the status messages to serve reads while satisfying sequential consistency.

In the design, we have decoupled the delivery of the actual updates and their metadata. The reason is that replicated storage systems usually have their own mechanisms in propagating updates among replicas. MeteorShower employs and wraps the native update propagation messages to *write notifications*. MeteorShower implements the periodic propagation of the metadata of the updates in *status messages*.

Write notifications are used to propagate the actual updates among replicas. Specifically, when a write is applied upon a replica, the update is also sent out to its peer replicas to eventually synchronize all the replicas. MeteorShower wraps the delivery of updates among replicas in write notification messages. A write notification is constructed using the following format:

$$WriteNotification = \{key, \quad update, \quad wts, \quad vn\}$$

It records the identification of the record (key), the updated value of the record ($update$), the request timestamp of the write (wts) and the version number of the record (vn).

For example, write notifications in Cassandra are the background data synchronization processes. They ensure that replicas are eventually consistent in the system. The updates among replicas are encapsulated in write notifications in MeteorShower. They not only synchronize data in replicas, but also wake up the pending reads in $readSubscriptionMap$.

A **status message** is an accumulation of the metadata of the updates conducted in an interval on one replica. The interval defines the frequency of exchanging status messages and it is configurable. A status message records the writes applied on a MeteorShower server using the following format.

$$StatusMessage = \{status, sts, seq, redundant\}$$

A status timestamp (sts) is included when the status message is sent out. It is the timestamp that we use to identify the staleness of a status message from replicas. A status message is sequenced (seq) using a universal MeteorShower server ID as the prefix. Thus, the sequence number allows recipients to group and order status messages with respect to senders. $redundant$ is a history of status messages in previous intervals. It is configurable, i.e., the number of previous status messages to be included, to tolerate status message lost. In order to keep the lightweight of status messages, replicas only send the metadata of the updates happened in the current interval instead of the metadata of the total storage namespace. The tradeoff is that our algorithm cannot tolerate the continuous l message loss during network transmission. l equals to the number of historical status messages in $redundant$. The $status$ is the accumulation of the metadata of updates as a list of (key, wts) happened during the current interval. At the end of each interval, a status message is propagated to all MeteorShower peers.

B. Implementation of MeteorShower

MeteorShower is completely decentralized. It is a peer to peer middleware, which is designed to be deployed on top of majority quorum based distributed storage systems. MeteorShwoer consists of six primary components as shown in Figure 8.

Status Message Dispatcher is the component that periodically sends out status messages to MeteorShower peers. A write received and processed by the write worker creates a metadata entry recorded in local status. Entries are aggregated to construct a status message when the dispatching interval is reached. A status message is sent out to all the MeteorShower peers every interval even when it is empty.

Message Receiver is a component that receives and processes write notifications and status messages. A write notification triggers the write worker to update the corresponding record to the underlying storage. A status message is used to update the status map, which decides whether a read request could be served. Both write notifications and status messages awake pending reads in the local read subscription map.

Status Map keeps track of status messages sent from MeteorShower peers. It is used to check whether a read request can be served with respect to the constraint of maximum staleness described in sub-section II.

Read Subscription Map maintains read requests when they cannot be served immediately because of lacking the required status messages or the updated data. The read subscription map is iterated when receiving new status messages or write notifications.

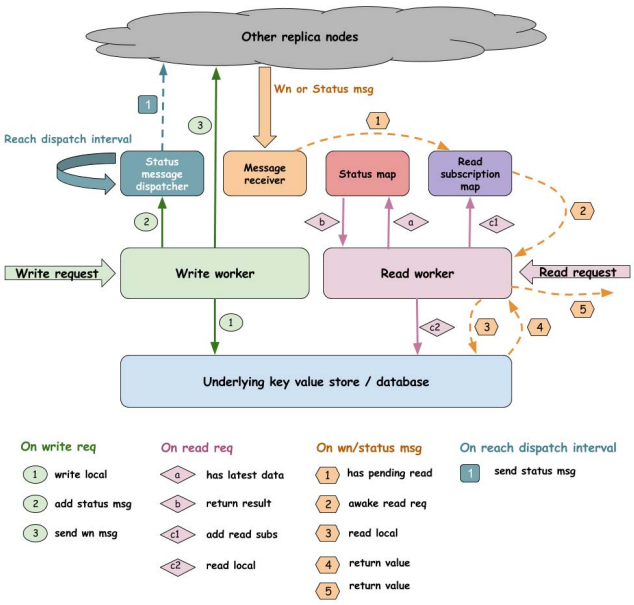

Fig. 8: Interaction between components

Write Worker persists record to the underlying storage if the update has a timestamp larger than the local timestamp of the affected data item. Then, it sends out write notifications about the update to its MeteorShower peers. In the meantime, an entry about this update is registered in the status message.

Read Worker decides whether the local data can be used to serve a read with the help of status messages. The read request is kept in the read subscription map when the local data cannot satisfy the maximum staleness constraint.

IV. Evaluation of MeteorShower

We evaluate the performance of MeteorShower on Google Cloud Platform(GCP). It enables us to deploy MeteorShower in a real geo-distributed data center setting. We first present the evaluation results of MeteorShower in a controlled environment, where we simulate multiple "data centers" inside one data center. It enables us to manipulate latency among different "data centers". Then, we evaluation MeteorShower with a real multiple data center setup in GCP.

1) MeteorShower on Cassandra: We have integrated MeteorShower with Cassandra, which is a widely applied distributed storage system. Specifically, we have integrated MeteorShower write worker, reader worker and message receiver components in the corresponding functions in Cassandra. Status message dispatcher, status map and read subscription map are implemented as standalone components. During our evaluation, we bundle the deployment of Cassandra and MeteorShower services together on the same VM. We adopt the proxy implementation in Cassandra, where the first node that receives a request acts as the proxy and coordinates the request.

2) The Baseline: We compare the performance of MeteorShower with the performance of Cassandra using different read/write APIs. Specifically, read "QUORUM" and "ALL" APIs are used as the baseline for read requests while write "ONE" and "QUORUM" APIs are employed as the baseline for write requests. The choice of these sets of APIs takes into the consideration of achieving sequential consistency. For example, the application of read "QUORUM" ("ALL") API together with write "QUORUM" ("ONE") API provides sequential consistency in Cassandra.

3) The Choice of θt in MeteorShower: We have implemented the MeteorShower algorithm with the lower bound and upper bound of θt, namely MeteorShower$_1$ and MeteorShower$_2$. The lower bound and upper bound of θt are presented in sub-section II. Specifically, in the case of read "QUORUM" ("ALL") operation in MeteorShower, it requires that the proxy server receives the status messages from a majority (all) of the replicas with a timestamp larger than $T - \theta t$. The write operations in MeteorShower are essentially the same as they are in Cassandra.

4) NTP setup: To reduce the time skew among MeteorShower servers, NTP servers are setup on each server. Briefly, NTP protocol does not modify system time arbitrarily. Time in each server still ticks monotonically. NTP minimizes the time differences among servers by changing the length of a time unit, e.g., the length of one second, in its provisioned server. We configure NTP servers to first synchronize within a data center, since the communication links observe less latency, which improves the accuracy of NTP protocol. Thus, there is one coordinator NTP server in a data center. Then, we have chosen a middle point, where observes the least latency to all the data centers, to host a global NTP coordinator. In this way, NTP servers inside one data center periodically synchronize with the local coordinator while local coordinators synchronize with the global coordinator. NTP is used to guarantee the bound of time drifts (ε) among servers. Empirically, we observe that NTP is able to keep the clock drifts of all servers within 1 ms most of the time. To be on the safe side, we evaluate our system under the maximum drift $\varepsilon = 2ms$.

5) Workload Benchmark: We use Yahoo! Cloud Serving Benchmark (YCSB) [4] to generate workload to MeteorShower. YCSB is an open source framework used to test the performance and scalability of distributed databases. It is implemented in Java and has excellent extensibility, where users can customize YCSB client interface to connect to their databases. YCSB provides a configuration file, using which users are able to manipulate the generated workload pattern, including read/write ratio, data record size, concurrent client thread number, and etc.

A. Evaluations in a Controlled Environment

In this experiment, we evaluate the performance of MeteorShower$_1$ and MeteorShower$_2$ under different cross data center network latencies in comparison with the original Cassandra baseline approach. Since latency cannot be manipulated in a real multi-DC setup, this experiment is conducted in a single data center with simulated multiple "data centers." Specifically, communications of VMs in different simulated "data centers" are introduced with a static latency. The latency is manipulated using NetEM tc commands [5].

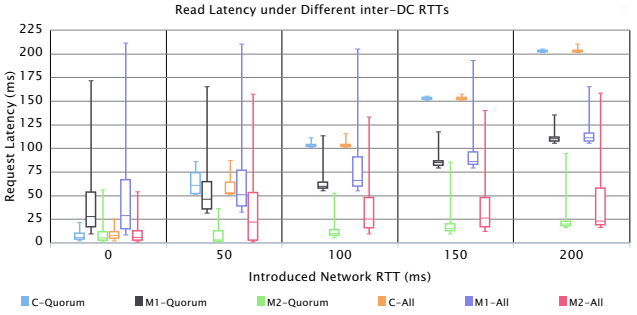

Fig. 9: Cassandra read latency using different APIs under manipulated network RTTs among DCs

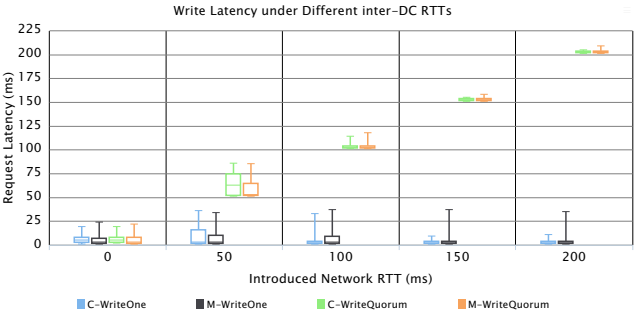

Fig. 10: Cassandra write latency using different APIs under manipulated network RTTs among DCs

For the cluster setup, we have initialized MeteorShower and Cassandra servers with the medium instances in GCP, which has two virtual cores. We have set up the cluster with 9 instances using 3 instances simulating a data center, which results in 3 data centers. We have spawned another 3 medium instances hosting YCSB in each simulated "data center". Each YCSB instance runs 24 client threads and connects to a local Cassandra/MeteorShower server to generate workloads. The composition of the workload is 50% reads and 50% updates.

Figure 9 and Figure 10 present the read and write latency of MeteorShower and Cassandra under different simulated cross data center delays, which are shown in the x-axis. We have run the workload with different combinations of read/write APIs in MeteorShower and Cassandra. Specifically, the workload is run against Cassandra, MeteorShower$_1$ and MeteorShower$_2$ with read QUORUM v.s. write Quorum and read ALL v.s. write ONE. We use write QUORUM instead of write ONE in combination with read ALL in the evaluation of MeteorShower$_2$, which enables it to use the upper bound of θt. The latency of each approach is aggregated from all the "data centers" and plotted as a boxplot.

Figure 9 shows that the latency of read QUORUM and read ALL operations in Cassandra, MeteorShower$_1$ and MeteorShower$_2$ are similar. This is because that "data centers" are equally distanced from each other, i.e., having the same network latency. Thus, waiting for a majority of replies requires more or less the same time as waiting for all the replies. As for Cassandra, the latency of its read operations increase with the

increase of network latency introduced among "data centers". The reason is that these operations need to actively request the updates from remote replicas before returning, which leads to a round trip latency. On the other hand, MeteorShower$_1$ only needs a single trip delay to complete read QUORUM/ALL operations in this case. This is because that a read at time t can be returned when it has received the status messages from a majority/all of the replicas with timestamp t. These status messages require a single trip latency to travel to the originator of the read plus the delay waiting for a status message dispatch interval of remote servers. MeteorShower$_1$ is not suitable to be deployed when the latency among data centers is less than 50 ms since it has non-negligible overhead in sending and receiving status messages. Despite the consumption of system resources, there is also a delay when waiting for a status message, which is sent every 10 ms in our experiment. It is reflected in Figure 9 when the introduced network latency is 0. Furthermore, this message exchanging overhead also causes a long tail in the latency of MeteorShower operations. Thus, MeteorShower is not suitable for applications that require stringent percentile latency guarantees. As for MeteorShower$_2$, which is configured with the upper bound of θt. It can be easily calculated that the upper bound of θt is a single trip latency introduced minus ε. It essentially allows read operations to return immediately since a read at time t only needs the status messages from a majority/all of the replicas with timestamp $t - \theta t$. Ideally, the status message with timestamp $t - \theta t$ should arrive at any "data center" no later than t plus a status message dispatch interval $10ms$. Thus, the latency of read QUORUM/ALL operations in MeteorShower$_2$ remain stable in the presence of increasing network latency among "data centers".

The writes of MeteorShower$_1$ and MeteorShower$_2$ are the same. So, we only show the writes of MeteorShower$_1$ in Figure 10. Since we have not changed the writes in MeteorShower comparing to the original implementation in Cassandra, the performance of both approaches is similar. However, we do observe a slightly long tail of write latency in MeteorShower, which is caused by the frequently exchanged status messages among servers in different "data centers".

B. Evaluations in Multiple Data Centers

We move on to evaluate the performance of MeteorShower in a multiple data center setup using GCP. We have used three data centers located in Europe, the U.S., and Asia as shown in Figure 11. The latencies between data centers are also marked in the figure. To make it consistent, we have used the same instance type and configuration as the previous experiments to set up the MeteorShower and the Cassandra cluster as well as the YCSB. We focus our evaluations on reads, since writes are processed identically in Cassandra and MeteorShower.

Figure 12 presents the aggregated read latency from the three data centers. As explained in the previous evaluation, the read requests of Cassandra experience a round trip latency. Specifically, read QUORUM operations experience the round trip latency from the closer remote data center while read ALL operations need to wait for the replies from the furthest remote data center. Read QUORUM/ALL operations in MeteorShower$_1$ observe a single trip latency from the closer/furthest remote data center. MeteorShower$_2$ performs

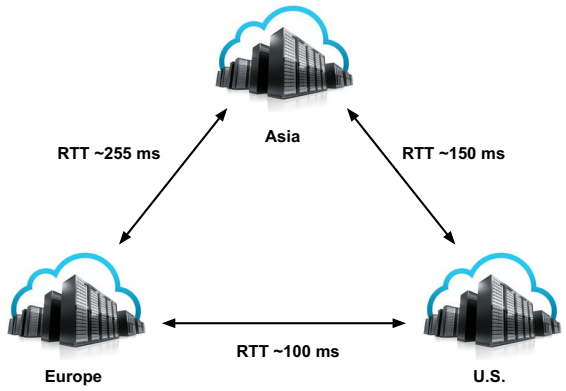

Fig. 11: Multiple data center setup

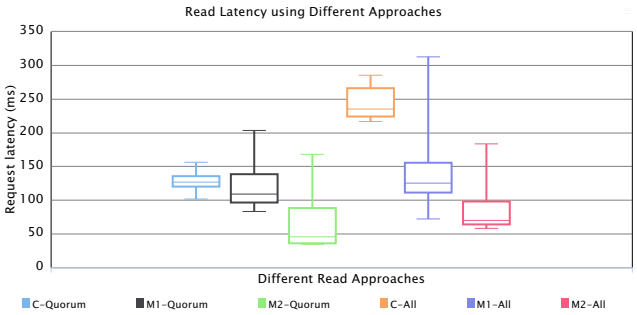

Fig. 12: Aggregated read latency from 3 data centers using different approaches

the best since it can use status messages that are θt earlier than MeteorShower$_1$. And in this setup, the upper bound of θt is around $50ms - \varepsilon$ for requests generated from Europe and U.S. and around $75ms - \varepsilon$ for requests originated in Asia.

More results regarding the read request latency in each data center are presented in Figure 13 and Figure 14. Specifically, Figure 13 shows the results grouped by different approaches while Figure 14 describes the latency grouped by data centers. We focus our explanation on the impact of different delays between data centers.

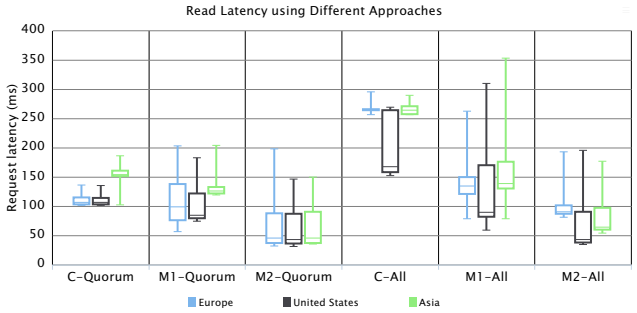

Fig. 13: Read latency from each data center using different APIs grouped by APIs

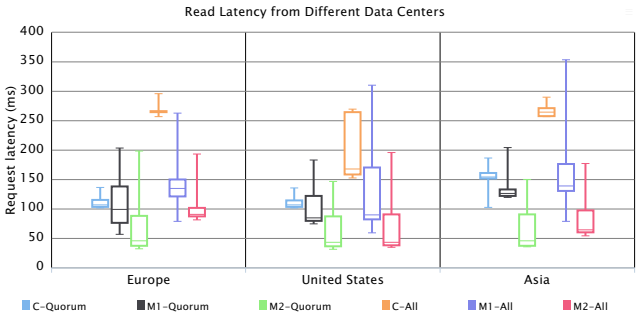

Fig. 14: Read latency from each data center using different APIs grouped by DCs

As we can see from Figure 13 and Figure 14, except MeteorShower$_2$, request latency in Asia is higher than request latency in Europe and U.S. This is because that the data center in Asia experience high latency to both Europe and U.S. data centers, especially Europe. On the other hand, MeteorShower$_2$ allows read QUORUM requests to return with the same latency as the requests in Europe and U.S. Specifically, MeteorShower$_2$ will expect a status message from U.S. data center instead of Europe, which is further. And a single trip communication from U.S. to Asia costs around $75ms$, which is compensated by a higher upper bound of θt of requests originated in Asia. Thus, the read QUORUM requests perform the same in Asia as the requests in Europe and U.S. even though Asia has the worst network connection. A similar conclusion can be drawn from the performance of read ALL, where requests perform even better in Asia than in Europe. Because requests from both data centers need to wait for status messages from the furthest data center. However, requests originated from Asia has a larger upper bound of θt ($75ms - 2ms$) than the requests initiated from Europe (upper bound of θt equals $50ms - 2ms$). So, the read ALL latency of requests in Asia is less than the request latency in Europe. As for MeteorShower$_1$, the performance of read ALL requests in Europe are similar to the performance of read ALL requests in Asia, since all requests need to pay for the highest latency and Asia data center does not have the advantage of a larger upper bound in θt. Obviously, the requests from the U.S. data center experience the least latency in all the cases. This is because that U.S. has the best connection to the other two data centers.

In sum, MeteorShower$_1$ needs a little more than single trip delay to return a read request, which is significantly faster than Cassandra in most of the requests. MeteorShower$_2$ is even faster than MeteorShower$_1$. It is able to return a read request immediately in most of the cases taken into account the reasonable overhead consumed to exchange status messages among data centers. Furthermore, MeteorShower$_2$ has a big advantage that it allows requests originated from a not well-connected data center (Asia) to be returned with improved latency. To some extent, the performance of MeteorShower$_2$ is irrelevant to the connectivity, in terms of latency, of a data center. Overall, the latency of MeteorShower has a longer tail than Cassandra, which makes it not suitable for percentile latency sensitive applications.

V. Related Works

Having a global knowledge of time helps to reduce the synchronization among replicas since operations can be naturally ordered based on global timestamps. However, synchronizing time in distributed systems is extremely challenging [6], which leads us to the application of loosely synchronized clocks, e.g., NTP [7]. Loosely synchronized clocks are applied in many recent works to build distributed storage systems that achieve different consistency models from casual consistency [8, 9, 10] to linearizability [11]. Specifically, GentleRain [8] uses loosely synchronized clocks to causally order operations, which eliminates the need for dependency check messages. Clock-SI [9] exploits loosely synchronized clocks to provide timestamps for snapshots and commit in partitioned data stores. Spanner [11] employs bounded clocks to execute transactions with reduced delays while maintaining the ACID property.

MeteorShower assumes a bounded loosely synchronized time on each server. It exploits the loosely synchronized time in a different manner. Specifically, a total order of write requests is produced using the loosely synchronized timestamp from each server. Then, read requests are judiciously served by choosing slightly stale values but satisfying the sequential consistency constraint. It is novel and different from the state of the art approaches, including Clock-SI, GentleRain, and Spanner, by allowing slightly stale values to be served in reads with respect to the global timeline.

Replicated logs are first proposed by G.T.Wuu et al. [12] to achieve data availability and consistency in an unreliable network. The concept of replicated log is still widely adopted in the design of modern distributed storage systems [13, 14, 15] or algorithms [16, 17]. For example, Megastore [13] applies replicated log to ensure that a replica can participate in a write quorum even as it recovers from previous outages. Helios [14] uses replicated log to perceive the status of remote nodes, based on which transactions are scheduled efficiently. Chubby [16] can be implemented using replicated logs as its message passing layer.

MeteorShower employs replicated logs for the similar reason: perceiving the status of remote replicas. However, MeteorShower exploits the information contained in the replicated logs differently. The information captured in the logs are the updates of replicas in remote MeteorShower servers. MeteorShower uses this information to construct a slight stale history of replicas stored in remote servers marked with loosely synchronized timestamps. Then, MeteorShower is able to judiciously serve requests with slightly stale values while preserving sequential data consistency, which significantly improves request latency.

The work of Egalitarian Paxos [18] provides performance improvement on Paxos operations when replicas are deployed in multiple data centers. EPaxos still requires at least a communication round to reach a decision. MeteorShower differs from EPaxos that, most of the times, it is able to smartly make a decision locally based on slightly stale information from status messages. A staleness bound is derived to achieve sequential consistency for majority quorum reads.

VI. Conclusions

MeteorShower is a novel read-write protocol for majority quorum based storage systems. It allows replica quorums to serve read requests more efficiently when replicas are deployed in multiple data centers. Essentially, MeteorShower exhausts the exploration of a global timeline, constructed using loosely synchronized clocks, in order to judiciously serve read requests under the requirement of sequential consistency. The algorithm allows MeteorShower to serve a read request without cross data center communication delays in most of the cases. As a result, MeteorShower achieves significantly less average and mean read latency comparing to Cassandra majority quorum operations. It is worth to mention that MeteorShower keeps all the desirable properties of a majority quorum-based system, such as fault tolerance, balanced load, etc. This is because that MeteorShower only augments the existing majority quorum-based operations. However, MeteorShower does observe some overhead. It scarifies the tail latency of requests because of the extensive exchanging of messages among remote replicas, which saturates the network resources to some extent.

A. Known Limitations

MeteorShower, like all the storage systems relying on majority quorums, does not tolerate the failure of a majority of servers. Furthermore, since MeteorShower extensively utilizes the network resources among servers, its performance depends on the network connectivity of those servers. It is observed in our evaluations that we have a significantly shorter tail in latency when MeteorShower uses an intra-DC network than an inter-DC network. The effect is more prominent under a more intensive workload. Thus, MeteorShower is not suitable for platforms where the performance of the network is limited. Lastly, the data consistency algorithm in MeteorShower relies on the physical clocks of all the servers. The correctness of MeteorShower depends on the assumption of bounded clocks, which means the clock of each server can be represented by the real-time clock within a bounded error. Thus, significant drifts in clocks interfere with the correctness and performance of MeteorShower. Luckily, there are techniques, such as the network timing protocol (NTP), to realize our assumption.

Acknowledgment

This work was supported by the Erasmus Mundus Joint Doctorate in distributed computing program funded by the EACEA of the European Commission under FPA 2012-0030 and the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research under the contract RIT10-0043. The authors would also like to thank the reviewers for their constructive comments and suggestions to improve the quality of the paper.

References

[1] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors (extended abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 679–690, New York, NY, USA, 1992. ACM.

[2] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[3] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[4] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[5] Stephen Hemminger et al. Network emulation with netem. In *Linux Conf Au*, 2005.

[6] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[7] NTP - The Network Time Protocol. http://www.ntp.org/. accessed: July 2016.

[8] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, New York, NY, USA, 2014. ACM.

[9] Jiaqing Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 173–184, Sept 2013.

[10] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[12] Gene T.J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.

[13] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[14] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. Minimizing commit latency of transactions in geo-replicated data stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1279–1294, New York, NY, USA, 2015. ACM.

[15] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[16] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[17] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *Conflict-Free Replicated Data Types*, pages 386–400. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[18] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.