Adaptive trade-off between consistency and performance for data replication

Hailong Sun, Bang Xiao, Xu Wang*,† and Xudong Liu

School of Computer Science and Engineering, Beihang University, Beijing, China

SUMMARY

Replication is widely adopted in modern Internet applications and distributed systems to improve the reliability and performance. Though maintaining the strong consistency among replicas can guarantee the correctness of application behaviors, however, it will affect the application performance at the same time because there is a well-known trade-off between consistency and performance. Many real-world applications favoring performance often choose to enforce weak consistency. Although there has been some work on flexible configuration of consistency, most focuses on design or deployment time. As the system settings constantly change during runtime, the tuning of the consistency-performance trade-off needs to be handled dynamically. Failing to do that will cause either underestimation or overestimation of the consistency and performance that can be achieved. Existing work does not well support the dynamic tuning of the aforementioned trade-off in runtime, which is mainly because of the lack of an appropriate quantitative model of consistency and performance. In this work, based on our previous effort on the quantitative model of consistency and latency, we design a replication protocol, CC-Paxos, to achieve an adaptive trade-off between consistency and performance according to application preferences and runtime information. By design, CC-Paxos is not bound to any specific underlying data stores. We have implemented CC-Paxos and applied it to MySQL databases. And real experiments both within a data center and across data centers show that CC-Paxos not only can dynamically adjust the delivered consistency in return for ensured performance but also outperforms MySQL Cluster in the case of strong consistency guarantee. Copyright © 2016 John Wiley & Sons, Ltd.

Received 13 January 2016; Revised 15 May 2016; Accepted 7 October 2016

KEY WORDS: distributed systems; data replication; consistency; Paxos; data management

1. INTRODUCTION

Nowadays, distributed databases have been widely deployed in practical application systems based on cloud computing [1] and big data [2]. In such applications, replication is often employed to achieve high availability and to avoid deteriorated performance caused by unbalanced load [3, 4]. But this incurs extra work to maintain data consistency across replicas as data can be added or modified on certain replicas. As a result, the introduced overhead can affect the system performance and availability. In this regard, CAP theorem [5] and PACELC [6] show that there exist two trade-offs in distributed replication systems: consistency and availability in the presence of network partition, and consistency and latency in the case of no network partition. Therefore, consistency is one important factor that influences the overall system performance like latency and availability.

In database management system area, there are two main camps: structured query language (SQL) and NoSQL. As data volume continuously grows, performance is becoming increasingly important for both of the two categories of database systems. For SQL databases, MySQL Cluster [7], VoltDB [8], and ScaleDB [9] are representative work on performance optimization, and they usually adopt...
strong consistency to support atomicity, consistency, isolation, durability (ACID) transaction at the expense of relatively low performance. Especially, MySQL Cluster, as one popular MySQL release, can only support two replicas at most [10]. It is no doubt that these SQL implementations are a limitation in terms of scalability. As for NoSQL systems, they often lower the consistency level (e.g., eventual consistency [11]) so as to obtain better performance and scalability (like BigTable [12] and Cassandra [13]). Because the consistency protocol is at the core of replication systems, its performance has a paramount impact on the overall performance of the system. In recent years, a lot of studies have been put forward to seek efficient consistency protocols. Some guarantee strong consistency like two-phase commit protocol (2PC) and replication state machine (RSM) [14]; others guarantee weak consistency that is widely used for high availability and low latency. For instance, eventual consistency [11] simply returns the latest version of data value stored locally, while causal consistency [15] returns the version of value that the other causally preceding versions will never be returned in future.

With the protocols mentioned above, most practical systems adopt a certain level of consistency, that is, either strong consistency or weak consistency. Nonetheless, real-world applications may ask for different levels of consistency according to application characteristics and performance considerations. Meanwhile, the performance difference among different consistencies can be tremendous. In recognition of the trade-off between consistency and performance, several storage systems [16–19] are providing multiple consistency options. But they do not well support the dynamic tuning of the aforementioned trade-off in runtime, because a major problem with existing multi-consistency storage systems is that they require applications complete the consistency configurations during deployment time or even development time. However, the workload, network speed, the degree of replication, and consistency requirement are varying in runtime [20, 21], which makes it hard to choose one specific consistency level for one application during deployment. Thus, it is necessary to adaptively tune consistency with the changing of system status. Lacking the adaptive mechanism can cause the pre-configured consistency/performance at deployment or development time to be either over optimistic or pessimistic, which accordingly results in failing to deliver the required consistency/performance or underestimating the system capability of guaranteeing consistency/performance. For instance, the ‘sold out’ messages unexpectedly prompted right before the final payment in e-commerce websites or the messy order of the reply messages to the same post in social network sites can often be observed, which is the result of sacrificing consistency for immediate response to user requests because latency is becoming an extremely critical performance factor that affects user experience [22]. However, such temporary data inconsistency should be avoided as best as possible if the latency of processing user requests can be controlled within a short period of time. Suppose latency is specified within a certain range (say [0.2s, 0.5s]); if the consistency can be adaptively adjusted in accordance with dynamic system states, user experience can be greatly improved, for example, in terms of less disappointing ‘sold out’ before checkout or less inconsistent order of reply messages.

In this work, we aim at relieving users from choosing the consistency configuration manually for each application and providing dynamic tuning of the trade-off between consistency and performance during runtime. In [23], we present a general model, RSM-d, to quantify consistency in replication systems. In this paper, we present CC-Paxos, a consistency configurable protocol for data replication. CC-Paxos is designed and implemented on the basis of RSM-d and the standard Paxos protocol, which can tune the trade-off between consistency and performance in consideration of application preferences and dynamic runtime information. Then we apply CC-Paxos to implement a replicated MySQL database, and we compare with MySQL cluster in an extensive set of real experiments.

Overall, we make the following contributions in this work:

1. We propose CC-Paxos, a replication protocol that can automatically tune system consistency and performance in runtime based on user preferences and system runtime information.
2. We have implemented a consistency tunable replicated database system with CC-Paxos on the basis of MySQL engine.
3. We conducted an extensive set of experiments in both a local cluster and between two clusters connected with Internet to evaluate the effectiveness and efficiency of our approach.
ADAPTIVE TRADE-OFF BETWEEN CONSISTENCY AND PERFORMANCE

The rest of the paper is organized as follows. In Section 2, we summarize the related work and introduce our previous work, which is an important working basis of this work. Section 3 presents the design of CC-Paxos. In Section 4, we describe the implementation of CC-Paxos on top of MySQL engine. Section 5 gives the experimental evaluation results, and Section 6 concludes our work.

2. RELATED WORK

Data consistency is a prominent problem for replication systems and has been receiving a lot of attention for many years. Early replication systems [24–26] pursue strong consistency through ad hoc methods or widely-used protocols. In recent years, a lot of research efforts have turned to understand the associated factors that affect consistency and varying consistency requirements in practical applications. To that end, the CAP theorem [5] and PACELC [6] state that it is not possible to obtain strong consistency, high availability, and partition tolerance at the same time. Traditional data stores often use 2PC [27–29] to manage all the atomic transaction participants to commit or abort a transaction to obtain strong consistency semantics while sacrificing availability, as shown in Figure 1(a). If and only if all the participants agree to commit a request, the coordinator will notify all participants to commit the request; otherwise, the coordinator will notify all participants to abort the transaction. As for RSM [14], another general approach to provide highly available services has three typical designs: nonuniform total order broadcast [30, 31], distributed consensus [32, 33], and uniform total order broadcast [30, 31]. The last two can guarantee strong consistency, while the first cannot. Paxos [32] is one famous RSM protocol of distributed consensus. For better performance, some data stores, like Taobao’s Oceanbase [34] and Microsoft’s SQL Server [35], choose Paxos to achieve strong consistency through guaranteeing that every replica process requests in the same order. Paxos is designed for the partially synchronous system model that requires \( n > 2f \) replicas to tolerate \( f \) crash failures. With Paxos, we do not have to wait for all replicas’ acknowledgments before committing or aborting a request. Given different values proposed by different replicas, only the value agreed on by a majority of replicas can be committed (Figure 1(b)).

Meanwhile, there are some efforts adopting weak consistency semantics so as to provide high availability when network partition happen. In this regard, eventual consistency [11] is a widely-used protocol. When there is a request for a data item identified with \( key \), any version of the data corresponding to the given \( key \) can possibly be returned. And clients are guaranteed to see the latest updated value eventually but with no guarantee of specific time. Another popular implementation is causal consistency [15], which can guarantee that any other value that causally happens before the returned value will not be returned in future.

Another body of research allows to configure the system consistency. For instance, Amazon’s SimpleDB and DynamoDB [16], PNUTS [17], Google AppEngine data store [18], and Oracle NoSQL Database [19] all support more choices of consistency for read operations. And users are required to determine the consistency before their applications are deployed. However, it is difficult to make the best decision of consistency configuration because the actual performance depends heavily on the factor of replication, locations of servers and clients, system workload, and other runtime information. To solve this problem, Terry et al. [36] present a consistency-based service level

![Figure 1. Two-phase commit protocol and Paxos.](image-url)
agreement system to provide six service level agreement (SLA) options for consistency configuration. Nonetheless, that work is tightly bound to a key-value store to implement those features and assigns specific execution flow for operation under each consistency levels. In contrast, our system with CC-Paxos can adaptively adjust specific quantitative consistency levels to meet application requirements for performance and consistency rather than directly configure discrete consistency levels. Meanwhile, CC-Paxos is designed for any data store, which is not bound to SQL or NoSQL systems.

Some recent work [37–39] provides solutions to implementing strong consistency with good performance and scalability in database systems. However, the solutions highly depend on underlying hardware design. For example, both [38] and [39] rely on Remote Direct Memory Access of the network hardware feature and/or Hierarchical Transactional Memory features to ensure the high performance data access, while Google Spanner [37] makes heavy use of hardware-assisted time synchronization by GPS clocks and atomic clocks to guarantee the consistency.

Our CC-Paxos use the results of our previous work [23] as the theoretical basis, in which we proposed a probabilistic method to quantify the consistency and latency of RSM-based systems. Given that RSMs are widely used in modelling replication systems and there are several specific RSM implementations, we proposed a general model, RSM-d, to give a unified description of commonly used replication protocols including nonuniform total order broadcast, distributed consensus, and uniform total order broadcast. In RSM-d, \( d \) represents the number of acknowledgement messages (ACKs) that must be received before committing a write. Typical protocols like eventual consistency, Paxos, and two-phase commit can be described with RSM-d by setting \( d \) to be 1, \([n/2] + 1\), and \( n \) respectively, where \( n \) is the total number of replicas. Essentially, \( d \) is a critical factor to determine the consistency and latency. Besides, the consistency is still subject to the probability that a write is lost or duplicated, and the latency is highly related to the message delay among replicas. For a given deployed system environment, all the factors can be measured by periodically sensing the network status. Therefore, the adjustment of consistency and latency can be achieved by adaptively tuning \( d \) according to the dynamically measured system parameters. With the RSM-d probability model, the consistency and latency of a system under a certain \( d \) can be quantitatively measured. Furthermore, combining the quantitative results of write inconsistency and latency of RSM-d, we provide a solution for trade-off between consistency and latency to achieve the maximum system benefit.

3. CC-PAXOS

In the previous section, we have seen that both 2PC and Paxos can guarantee the strong consistency. The difference is that 2PC needs \( n \) ACKs, while Paxos only needs \([n/2] + 1\) ACKs before committing a request. Because both of them are designed for guaranteeing strong consistency, thus, they are not suitable for the situation where performance is preferred to consistency, such as the social network and e-commerce applications described in Section 1. In this section, we present CC-Paxos, a new protocol for dynamically adjusting the delivered consistency and performance according to application preference and runtime state information.

CC-Paxos is designed on the basis of our RSM-d model and Paxos. On the one hand, CC-Paxos leverages the quantitative model defined in RSM-d to tune the parameters that affect system consistency and performance. On the other hand, CC-paxos adopts Paxos protocol as the basis of implementing the basic consistency enforcement mechanism. Because CC-Paxos is a variation of Paxos, it can be described with a group of roles including proposer, acceptor, and learner. After receiving a write request from clients, a proposer will broadcast the new value to all acceptors. As a proposer, it depends on a majority of acceptors to agree on a value it proposes. And the acceptors decide whether to accept the proposed value by sending an ACK back or not. Once a value is agreed, the learners can learn it. In comparison with Paxos, the proposer does not have to receive \([n/2] + 1\) ACKs to commit the write. The number of ACKs to trigger commit operation is denoted as \( d \), which can be any value in \([1, [n/2] + 1]\). As the value of \( d \) varies, the consistency that CC-Paxos delivers will change accordingly.
3.1. Normal operation

Algorithm 1 presents CC-Paxos; we describe it as three phases.

Phase 0 (Task 1) is the preparation phase. The leader (a chosen node that coordinate the whole phases) will first use Equations (1) and (2) provided in our preliminary work [23] to compute the consistency and latency. To do that, it is necessary to collect the required parameters, which will be explained in Section 4. After that, following applications’ preference for consistency and latency, a particular $d$ will be computed with the consistency configuration algorithm that will be described in Section 4.

\[
P_{wc} = P_{wl} + P_{wd} = \sum_{m=d}^{[n/2]} P_{elw}(m)Q(m) + (1 - P_c)P_{cfg}P_{no}
\]

where $P_{wl}$ and $P_{wd}$ stand for the probability of lost write and write duplication, respectively.

\[
E(L_w(d + 1) - L_w(d)) = \left(\frac{n-1}{d-1}\right)\int_{0}^{+\infty} (G(t))^{d-1}(1 - G(t))^{n-d}dt (d \geq 1)
\]

where $G(t)$ is a cumulative distribution function of $f(t)$ that denotes the probability density function of the message delay between the coordinator and the replica. To compute with Equation (2), we define some symbols as shown in Table I. If $d = 1$, $L_w(1) = t_{log(coordinator)} + min(t_c(i) + t_{com}(i) + t_f(i))$.

Phase 1 (Tasks 2 and 3) is similar to the second phase of Paxos. As we can see from lines 12–16, when the leader starts Phase 1, it generates a unique id ($cid$) for the value ($v$) from proposer and sends $cid$ to all the acceptors as Phase 1A message. An acceptor will reply with the greatest id ($rid$) that it has seen, the greatest id ($rid_r$) it has proposed, and the corresponding value ($val_r$), or NULL if it has not accepted any proposal as Phase 1B message. In the event that the received $cid$ is smaller than $rid_r$, the received prepare message will be disregarded (lines 18–22).

In Phase 2 (Tasks 4–6), the leader will choose a value ($val_c$) based on the messages received from acceptors; then it commits its proposal for this value with its unique identifier ($vid_c$) and $cid$ as Phase 2A message to all acceptors (lines 24–34). Different from original Paxos that requires the proposer to receive ACKs from a majority of acceptors, CC-Paxos allows to start Phase 2 after it receives a particular $d$ (which can be any value between 1 and $[n/2] + 1$) ACKs from acceptors. The value of $d$ is determined in Phase 0. Meanwhile, if $d$ is equal to $[n/2] + 1$ or greater, CC-Paxos provides strong consistency; if $1 \leq d < [n/2] + 1$, CC-Paxos will provide weak consistency. The acceptors again answer with Phase 2B message only if they have not involved in any higher numbered instance (lines 36–40). At the same time, acceptors will inform all the learners with the value they have accepted. A learner will not decide any value until it receives $d$ messages from acceptors for the same instance. At last, the leader will forward SUCCESS message to the proposer as shown in Task 6.

CC-Paxos not only provides strong consistency but also supports weak consistency. Benefitting from the less ACKs required to commit a proposal than original Paxos, CC-Paxos can improve the performance. In Section 5, we will verify this through experiment results.

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{log}(i)$</td>
<td>the consumed time of logging a write for replica $i$</td>
</tr>
<tr>
<td>$t_c(i)$</td>
<td>the consumed time of the COMMIT message from the coordinator to replica $i$</td>
</tr>
<tr>
<td>$t_{com}(i)$</td>
<td>the consumed time of committing a write for replica $i$</td>
</tr>
<tr>
<td>$t_f(i)$</td>
<td>the consumed time of the FINISHED message from replica $i$ to the coordinator</td>
</tr>
</tbody>
</table>
Algorithm 1 CC-Paxos

1: **Parameters**
2: \( \delta \)  \( \triangleright \) delay for reconfigure the consistency level

3: **Initialization**
4: \( p_c \leftarrow 0 \)  \( \triangleright \) crash probability of all replicas
5: \( \text{latency} \leftarrow 0 \)  \( \triangleright \) latency between replicas
6: \( \text{accepted} \leftarrow 0 \)  \( \triangleright \) the set of replicas with the same status

7: **Task 1 (leader)**
8: \text{Every } \delta \text{ time do}
9: compute \( \text{latency} \) and \( p_c \)
10: \( d \leftarrow \text{setMajority}(\text{latency}, p_c) \)

11: **Task 2 (leader)**
12: \text{Upon receiving value } v \text{ from proposer}
13: increase \( cid \) to an arbitrary unique value
14: for all replicas do
15: send(replicas, (\text{Phase1A}, cid))
16: end for

17: **Task 3 (acceptor)**
18: \text{Upon receiving (Phase1A, cid) from leader}
19: if \( cid > rid \) then
20: let \( rid \leftarrow cid \)
21: send(leader,(\text{Phase1B}, rid, rid_r, val_r))
22: end if

23: **Task 4 (leader)**
24: \text{Upon receiving (Phase1B, rid, rid_r, val_r) from replica } p
25: select the largest \( rid_r \) value received labeled as \( k \),
26: \( \text{accepted} \leftarrow \text{accepted} \cup p \) where \( rid_r = k \)
27: \text{upon accepted contains } d \text{ items}
28: if \( k = 0 \) then
29: let \( val_c = v \)
30: else
31: let \( val_c = val_r \)
32: end if
33: \( \text{vid}_c \), identifier for \( val_c \)
34: send (replicas, (Phase2A, cid, \text{vid}_c, val_c))

35: **Task 5 (acceptor)**
36: \text{Upon receiving (Phase2A, cid, vid_c, val_c) from leader}
37: if \( cid \geq \text{rid} \) then
38: \( \text{rid} \leftarrow \text{cid} \& \text{val}_r \leftarrow \text{val}_c \& \text{rid}_r \leftarrow \text{vid}_c \)
39: send(leader, (\text{Phase2B}, cid))
40: end if

41: **Task 6 (leader)**
42: \text{Upon leader receive } d \text{ Phase2B messages}
43: forward \text{SUCCESS} \text{ message to proposer}
3.2. Handling failure

Because messages between replicas can be lost in asynchronous system communication, we will retransmit the lost messages. If the leader does not receive any reply to its Phase 1A and Phase 2A messages, it will send them again until it receives a response or speculates that certain acceptors have crashed after a certain period of time. Once a failed acceptor is detected by the leader, the recovery mechanism will notify the proposer to re-execute Phase 1. As for acceptors, they store their states in persistent storage before sending Phase 1A and Phase 2A messages so as to ensure the state information is not lost.

In CC-Paxos, although all the nodes are mostly stable, there is still a probability of node crashing. We have discussed the situation of acceptor crash, and we further analyze how to handle leader crash. Once a leader crash is detected, a new leader will be chosen. In the event a crashed leader recovers while a new leader has been chosen in the mean time, there may simultaneously exist multiple leaders. When this happens, Paxos can still guarantee system safety by comparing the last ID that they have accepted. While in CC-Paxos, due to the flexibility of consistency configuration, we cannot handle this situation by just comparing the last accepted ID because they can be the same. So when a leader or any replica restarts, it will recover their status based on local historical records, change its role to be a learner, and then catch up the decision of instance during its crash time.

3.3. Optimization

In CC-Paxos, in Phase 0, we will detect some system parameters to dynamically configure the consistency level. If all this detecting work is conducted by the leader, it can potentially cause performance bottleneck because the leader is the centralized point. To relieve the workload of a leader, we disseminate the detection workload to all replicas, let each replica capture its status, and push it to the leader. Generally, we detect the latency between each replica at longer intervals in LAN settings than in WAN environments.

Moreover, in high-latency networks, it is too long for the leader to wait for the Phase 2B messages if the leader executes instances one by one. Therefore, we implement the following optimization method. The leader uses pipelining to execute a number of consensus instances in parallel [40]. In Phase 2, instead of executing a single value per instance, it executes a batch of proposed values for a single instance [41].

4. IMPLEMENTATION

In this section, we describe the implementation of a replicated DB system that can automatically adjust the consistency according to the configured application preference and dynamic runtime information. In our implementation, we adopt MySQL to be the underlying database engine and each data replica is located in one MySQL node. Note our work can be easily implemented on top of other database engines. Data operations including SQL queries or NoSQL’s get/put are all considered as requests, and CC-Paxos can guarantee the requests are processed with certain order by underlying database engines.

4.1. Architecture

As shown in Figure 2, the prototype system mainly consists of CC-Paxos Client, CC-Paxos Server, and underlying MySQL database engines.

**CC-Paxos Client.** This is a light-weight client library that an application uses to read/write data from the back-end replicated database engines. For the write requests, the client will forward the requests to the leader replica node. If it cannot connect to the leader node, it will initiate a leader election process, which is the same as Paxos [42], so as to find a new leader. For read requests, the client will forward the reads to a random chosen replica, which helps to balance the load among replicas.
CC-Paxos Server. This module serves as an agent for each database engine instance; thus, there is one CC-Paxos Server deployed on each MySQL node. Each server instance can handle the received read requests, while only the leader node can receive and deal with write requests from the client. CC-Paxos is employed to coordinate the servers to deliver the required consistency. When write requests arrive, CC-Paxos is responsible for determining the sequential order of the coming requests and further ensuring the requests are executed in the same order on each replica. Finally, CC-Paxos Server will execute the write on the corresponding database engine. To fulfill the above functions, the following sub-modules are designed and implemented.

Request Manager. This sub-module is responsible for listening connection, receiving requests from clients, translating the requests to CC-Paxos sub-module, and finally, sending the answers back to the Client. Once the connection between the Client and Client Manager is established with the first request, it will be cached so as to avoid the delay of connection establishment for later requests. If the connection is broken, for example, the server is down for some reason, the client library will try to reconnect to the leader within a specified time-out. CC-Paxos will select a new leader if the crashed server recovers. After that, the client will connect to a new selected leader and then send requests. After a request is received, it will be put into a request queue. When the request queue is full, all the requests will be dequeued and passed to CC-Paxos module to get ordered. This batch mechanism is commonly used in distributed systems [43] to improve general throughput and average request processing. This is because reaching distributed consensus with CC-Paxos incurs extra overhead and processing the requests in batch can greatly reduce the overhead compared with processing requests individually. We also use a time-based trigger: a batch is started whether it is full or not, if it has been waiting for a configured time-out. In our prototype system, the default of the batch size and the time-out are 64 KB and 10 ms, respectively. After CC-Paxos determines the order, this module will send the requests to MySQL for processing.

CC-Paxos. This sub-module is responsible for establishing a total order among the client requests using the CC-Paxos protocol, which is described in Section 3.

Replica Manager. It mainly deals with the sending and receiving of messages among replicas during the execution of CC-Paxos protocol. Meanwhile, it also periodically detects the latency among replicas and then send the information to Consistency Configurator module.

Consistency Configurator. It reads the application preference information, receives runtime information, and use those information to compute the specific consistency level periodically using the probabilistic model described [23].

DB Connector. This module is responsible for connecting to MySQL via Java database connectivity (JDBC). It receives the requests that are determined in CC-Paxos Server and sends them to MySQL to execute.

4.2. Determining the consistency level

As we described in Section 3, CC-Paxos needs some system parameters to reconfigure the consistency levels according to the RSM-d model. First, we need to collect the system parameters for RSM-d model. Second, we design an algorithm to determine $d$ based on RSM-d model.
4.2.1. Detecting the parameters. CC-Paxos needs both static configuration and dynamic system status to compute the consistency level with Equations (1) and (2).

Static parameters: (i) $FDSuspectTimeout$ represents the time to wait before confirming that the leader fails and (ii) $FDSendTimeout$ represents the interval to send heartbeats. Both of the two parameters are configured in a file, and they are used to compute $P_{fs}$.

Dynamic parameters are captured at runtime including $P_c$, $T_c(i)$ and $T_f(i)$. And $P_c$ stands for the crash probability of replicas. In our implementation, we mainly consider the crashes because of network failure. Thus, we measure two cases that replicas will encounter crash to compute $P_c$: (i) $P_{c1}$, the probability of connection failure among the leader and the replicas; and (ii) $P_{c2}$, the probability of DB connection failure. For $P_{c1}$, we use a thread to capture the time-out information when the current node is trying to connect with other replicas. To control the extra overhead, it is necessary to reduce the number of detection messages within the overall network. Therefore, we set an interval that can be configured in a file to send the detection messages. We count the number of timeout messages $n_{out}$ after $n_1$ times of connection attempts and then describe $P_{c1}$ as: $P_{c1} = n_{out}/n_1$.

Similarly, for $P_{c2}$, we set a thread to periodically connect to the database engine. After the thread makes $n_2$ times of connection requests, the connection failure $n_f$ is recorded. We have $P_{c2} = n_f/n_2$.

In most cases, network and MySQL connection failures are mutually exclusive events. Thus we have

$$P_c = P_{c1} + P_2 \quad (3)$$

To use Equation (2), parameters ($T_c(i)$ and $T_f(i)$) are necessary. $T_c(i)$ stands for the latency of sending the COMMIT message from the coordinator to replica $i$, and $T_f(i)$ refers to the latency of sending the FINISHED message from replica $i$ to the coordinator. In our implementation, we use the averaged latency between the coordinator and a replica as the value for both $T_c(i)$ and $T_f(i)$, which is measured by sending messages periodically.

4.2.2. Consistency configuration algorithm. As shown in Algorithm 2, CC-Paxos computes a suitable consistency level that satisfies application preferences during runtime. First, users are expected to propose their demands for consistency degree ($1 - UserPwc$) or just specify strong/weak consistency and latency ($UserLatency$) that stands for the lowest consistency level and the highest latency that the application can accept.

The algorithm starts with user demands and then computes the system consistency level $1 - P_{wc}$ and response time latency under different $d$ from 1 to $[n/2] + 1$, by comparing the $P_{wc}$ and latency under different $d$ from 1 to $[n/2] + 1$ with user demands; it then selects the best $d$ that conforms the consistency and latency level. When there are more than one value meeting the demands, we will evaluate whether user prefers to consistency demand or latency demand; this helps us make the final choice.

In Line 12, we set majority as $[n/2] + 1$, in consideration of that when the user put forward some strict requirements that we cannot conform to all of them, we should guarantee the consistency to provide strong consistency service.

4.3. Read and write operation

In our implementation, the client is configured with the IP addresses or URLs of replica nodes. When the client tries to send the first request, it will randomly choose a replica to connect. In case of failure, it will keep trying another replica until a successful connection is established. If the request is a write request, the request will be redirected to the leader node for further processing. As described before, a write request will be put into a queue and then sent to CC-Paxos together with other requests to determine a sequential order. Finally, it will be executed in the specified order.

For a read request, it can be processed by any replica locally. However, this does not guarantee the freshness of the data. If the application needs to read the most recent data, this can be implemented by redirecting read requests to the leader node.
Algorithm 2 consistency configuration algorithm

1: procedure SET MAJORITY(latency$_i$, $p_c$)
2:     initialization $d = 1$
3:     loop until $d > \lfloor n/2 \rfloor$
4:         Compute the system consistency level $P_{wc}$ under a specific $d$
5:         Compute the system latency under a specific $d$
6:         if $P_{wc}$ and latency satisfy user demands then
7:             Store this $d$ temporarily in $tems$
8:         end if
9:         $d = d + 1$
10:     end loop
11:     if $tems$ is empty then
12:         $majority = \lfloor n/2 \rfloor + 1$
13:     else if user prefer to Latency then
14:         $majority = temps[0]$
15:     else if Latency of $d = \lfloor n/2 \rfloor + 1 < UserLatency$ then
16:         $majority = \lfloor n/2 \rfloor + 1$
17:     else
18:         $majority = \text{last value of } temps$
19:     end if
20: end procedure

5. EXPERIMENTAL EVALUATION

In this section, we present our evaluation results of how CC-Paxos can help achieve good performance and meet the specified consistency in the meanwhile. We mainly focus on measuring the metrics of throughput and latency under different values of the request size and the number of replicas. And we revise ran CC-Paxos and Paxos, respectively, on MySQL. At the same time, we compared our implementation of CC-Paxos on MySQL with MySQL Cluster.

The experiments were run in a data center and across two data centers, respectively. For the first case, we used a cluster in LAN consisting of seven physical machines, each of which was equipped with a 8-core 2.5-GHZ CPU and 50-GB RAM and was connected to each other with 10-Gbps Ethernet. And the clients were in the same LAN. For the second case, we used two virtual clusters rented from Aliyun, one of which was composed of two nodes in QingDao City and the other of which was composed of two nodes in HangZhou City. Each node has a 2-core 2.2-GHz CPU and 200-MB RAM, and there was a 1-Gbps Ethernet within a data center while a 20-Mbps Ethernet between two data centers. The clients were set up in the same LAN with the cluster in QingDao City. Furthermore, all the throughput were the maximum throughput obtained by increasing the number of subsequent clients with 50 threads per client to send requests.

5.1. Consistency choices

In this experiment, we evaluate the effectiveness of CC-Paxos’s consistency tuning according to application preferences. As shown in Table II, we consider four application preferences that can be configured by application managers. Each preference configuration describes whether consistency or latency is preferred. For example, consistency (2.5) means strong consistency should be enforced and a latency of 2.5 ms is expected, while latency (2.5) indicates a latency of 2.5 ms should be guaranteed, while consistency does not have to be strong. The experiments are run in the local cluster. Each data item is 4 bytes and has three replicas. The results of consistency, latency, and the value of $d$ are shown in Table II. We can observe in both Cases 2 and 4 that CC-Paxos achieves a stronger consistency level with $d = 2$. While for Cases 1 and 3, because latency is preferred, CC-Paxos guarantees the latency requirement while sacrifices the consistency with $d = 1$. These results are consistent with Algorithm 2 described in Section 4.2.2.
Next, we run experiments by varying the number of replicas so as to evaluate how the performance is affected by different consistency levels through changing $d$. In Figure 3, we show the throughput of CC-Paxos, while $n = 3$, $n = 5$, and $n = 7$ with different values of $d$. The data size is also 4 bytes.

Generally, we can observe that as $d$ increases, the throughput of CC-Paxos decreases accordingly because of the overhead of synchronization among replicas. For instance, when $n = 3$, $d$ can take two values: 1 and 2. The throughput of CC-Paxos is around 5200 with $d = 1$ and 4900 with $d = 2$. CC-Paxos with $d = 1$ achieves 4.9% improvement of the throughput compared with strong consistency. The improvement increases to 6.4% when $n = 7$.

With the above experimental results, we show that CC-Paxos is able to adapt the delivered consistency and performance according to the configured application preferences.

5.2. Throughput comparison in LAN

As CC-Paxos is responsible for handling write requests, we evaluate its performance through measuring its throughput with different data size of writes and number of replicas. In the meantime, we compare with MySQL Cluster 7.3.6 and Paxos implementation on top of MySQL, which all enforce strong consistency among data replicas.

As shown in Figure 4(a), when the size of written data increases, the throughput of both MySQL Cluster and CC-Paxos decreases because a request for more data costs more network overhead. The peak throughput of CC-Paxos reaches 5400 requests per second, while that of MySQL Cluster is about 4900 requests per second. The reason is that MySQL Cluster adopts 2PC to ensure the consistency among replicas, which induces more network overhead than CC-Paxos. To show the performance advantage of CC-Paxos over the 2PC used by MySQL Cluster, we plot the ratio of their throughput. It can be shown that as data size increases, the advantage of CC-Paxos is getting greater. Figure 4(b) plots the throughput of MySQL Cluster and CC-Paxos when the number of concurrent clients grows, where the size of written data in a request is set to be 4 bytes. It can be observed that CC-Paxos obtains the largest throughput of 5.4k requests per second when the number of concurrent clients reaches 1500. In contrast, MySQL Cluster achieves the maximum throughput of 4900 requests per second when the number of concurrent clients is just 300. This means that
CC-Paxos has greater scalability than MySQL Cluster because of the optimization mechanisms of CC-Paxos with pipelining and batch processing.

As MySQL Cluster can only support two replicas at most, Figure 4 only show the results when there are two replicas. Next we vary the number of replicas as 2, 3, 5, and 7, respectively, and compare the throughput of CC-Paxos and Paxos under different size of written data. The data size are set to be 4 bytes, 40 bytes, 400 bytes, and 4000 bytes, respectively. Figure 5 shows that the throughput of both CC-Paxos and Paxos drops as more replicas are used, which is the result of increasing synchronization overhead when the number of replicas grows. We can also observe that CC-Paxos consistently outperforms Paxos in all the cases, and its advantage over Paxos gets bigger as the size of written data increases. This is because CC-Paxos does not keep a strong consistency write all the time based on the application preferences. From Figure 5, we can see that the performance of CC-Paxos is slightly lower than Paxos when they both guarantee strong consistency, because of more message transmission and resources occupation for monitoring system status. But we gain greater performance improvement when CC-Paxos sacrifices strong consistency.

Figure 4. Throughput of MySQL Cluster and CC-Paxos $d = 1$.

Figure 5. Throughput of CC-Paxos and Paxos in different size of requests.
5.3. Throughput comparison in WAN

In the following, we present experimental results in a cross-data center scenario, in which two clusters from Aliyun.com are located in Qingdao and Hangzhou, respectively. For simplicity, we denote the two clusters as Cluster-Q and Cluster-H, respectively. Data replicas are distributed across the two clusters. In such an environment, network latency among replicas is much larger than the case of within-a-data center.

Figure 6(a)–(c) show the throughput of CC-Paxos and Paxos as the size of written data grows when the number of replicas is 2, 3, and 4, respectively. When $n = 2$, one replica is in Cluster-Q and the other is in Cluster-H; when $n = 3$, two replicas are in Cluster-H and one is in Cluster-Q; when $n = 4$, the four replicas are evenly distributed across the two clusters. Figure 6(a) plots the results of $n = 2$; we can see that CC-Paxos gets 32% higher throughput than Paxos when the request size is 4 bytes, and this advantage increases to about 44% as the request size reaches 4000 bytes. Figure 6(b) and (c) shows the results with $n = 3$ and $n = 4$, and the trend is just like the case of $n = 2$. But the advantage of CC-Paxos over Paxos is much larger when $n = 4$, while it is relatively small when $n = 3$. Furthermore, when the request size is 4 KB, the throughput of Paxos with $n = 3$ is larger than the one with $n = 2$ (14,600 vs. 13,400). The reason is that although for both of the two cases Paxos needs to wait for two ACKs before a request can be committed, for the case of $n = 3$, the first two ACKs are from the same cluster, which avoids the cross-cluster communications that the case of $n = 2$ must do. As for CC-Paxos, the throughput decreases with the number of replicas but is still greater than the one of Paxos. This is mainly because the ACKs are from the same cluster and CC-Paxos does not need $\lceil n/2 \rceil + 1$ ACKs all the time.

5.4. The fluctuation of network

The former experiments show that the CC-Paxos has better performance compared with Paxos either in LAN or WAN. The reason is that it is not necessary for CC-Paxos to receive $\lceil n/2 \rceil + 1$ ACKs to commit requests. In this experiment, we show that CC-Paxos also has a stronger robustness when it encounters network fluctuation. We use a cluster with five nodes, and the request size is 4 bytes. As for application preferences, we set it as latency (2.5). Furthermore, we use exponential distribution with a rate $\lambda$, where $\lambda$ is 1, to simulate the fluctuation of network latency. It means that we set additional latency ([1,2] ms) when we transmit requests and messages. Each experiment lasts for 150 min, with a fluctuation being triggered 50 min after start-up, and we compute the throughput and response time every 5 min.

Figure 7(a) shows the results of response time. Before the fluctuation being triggered, the response time of Paxos and CC-Paxos stays steady for 50 min after start-up. However, after the network latency slows down, the response time of Paxos is observed at 1 ms increase, while the response time of CC-Paxos just increases by around 0.2 ms.

Figure 7(b) shows the throughput of Paxos and CC-Paxos lasts steadily before the injected network fluctuation. Then the throughput of CC-Paxos slightly decreases to around 41,000 requests per second (compared with 49,400 before the fluctuation), however the throughput of Paxos sharply decreases to around 35,000 requests per second (compared with 49,000 before).

The reason is that at the first stage (the 50 min after start-up), according to the preference of application (latency(2.5)) and system status, CC-Paxos sets the $d$ as 2, which means the leader just
needs two ACKs to commit a request. When it comes to the second stage, the latency between replicas becomes larger; at this moment, CC-Paxos reconfigures the value of $d$ to 1 to satisfy the preference of application. This reconfiguration makes the decrease of throughput of CC-Paxos and the increase of response time much smaller than those of Paxos.

6. CONCLUSION

Data replication is commonly used to improve system performance. Apart from the benefits, replication also brings in additional overhead for maintaining the consistency among replicas. In practice, not all applications require strong consistency, and they can tolerate temporary consistency for better performance. In this work, we design CC-Paxos, a replication protocol on the basis of our previous quantitative consistency model and Paxos protocol. With CC-Paxos, application users can specify their preferences in terms of consistency and performance, and the concrete consistency level will be computed and adjusted during runtime. We have implemented CC-Paxos and applied it to MySQL database engine. Experimental results demonstrate the effectiveness of our work.

In the future, we will improve this work from two aspects. First, we will experiment CC-Paxos to other database engines to test its generality. Second, we will improve the runtime parameter detection mechanisms for the better tuning of consistency and performance.

ACKNOWLEDGEMENTS

This work was supported partly by National Key Research and Development Program of China (no. 2016YFB1000804), National Natural Science Foundation of China (no. 61370057 and no. 61421003) and partly by China 863 program (no. 2015AA01A202).
ADAPTIVE TRADE-OFF BETWEEN CONSISTENCY AND PERFORMANCE

REFERENCES
