2016 IEEE 22nd International Conference on Parallel and Distributed Systems

CC-Paxos: Integrating Consistency and Reliability in Wide-Area Storage Systems

Yili Gong, Chuang Hu, Wentao Ma
Computer School, Wuhan University
Wuhan, Hubei, China
Email: {yiligong, huchuang, mawentao}@whu.edu.cn

Wenjie Wang
EECS, University of Michigan
Ann Arbor, MI, USA
Email: wenjiew@umich.edu

***Abstract*—Data replication is widely used in geo-distributed storage systems, and strong consistency is preferred for correctness and programming simplicity at the application layer. To address the inefficiency and insufficiency of the causal consistency model, a strong consistency model named distributed context consistency is defined. It explicitly defines the necessary dependencies among distributed clients to effectively reduce false-positive dependencies among operations. A consensus algorithm named CC-Paxos is proposed to implement this distributed context consistency model. It exploits timestamps for operation sequencing in distributed contexts and adopts fine-granularity dependency checking to effectively reduce the number of potential conflicts. Experimental results show that, compared with implementations using causal+ consistency model in the upper layer and Egalitarian Paxos in system layer, CC-Paxos can significantly decreases latency and increases throughput with no sacrifice on scalability.**

***Index Terms*—Distributed Context; Distributed Context Consistency; Consensus; CC-Paxos**

I. Introduction

Geo-distributed data storage is widely deployed in both public and private cloud platforms, where data replication across sites is commonly used for high system throughput, low latency, and high service reliability. Even though some Internet services [1], [2] adopt the eventual consistency model for scalability, applications appreciate stronger consistency for its simplicity. The weaker consistency provided by the storage system generally requires service providers or application developers to implement and guarantee the strong semantics for application correctness and user experience. Such efforts are usually laborious and error prone. Therefore, system designers seek the strongest consistency model that they can provide under the practical constraints, such as causal+ consistency [3] and real-time causal consistency [4].

There are two issues in the original causal consistency model and its variants. First, operations in a single thread of execution are considered to be in potential causality, and this causal relationship is transitive. Such consistency definition forces unintentional order requirements to operations and abates concurrency of operation execution. Second, the original model lacks capability for applications to specify ordering between two operations. To address these two issues, this paper redefines the concept of *context*, which was originally introduced as the "thread of execution" and used to track causal dependencies among a client's operations [5]. The new concept is called *distributed context*, where the execution environment accommodates one or more clients, from which operations could be issued.

Based on this new definition, we further propose a strong consistency model named *distributed context consistency*, which allows clients on different sites can share the same context, and the sequence of operations in the same context will be enforced. With this consistency model, developers can avoid unnecessary operational dependencies and explicitly define the required dependencies in separate clients easily. The lower-level storage system will ensure the operation sequence as specified.

System performance is another critical metric for a well-designed consistent model. To achieve high reliability with replication, consensus algorithms [6], [7] are commonly adopted for fault-tolerance. Egalitarian Paxos [8](EPaxos) based on Paxos is a distributed consensus algorithm without master node. Exploiting conflict-checking mechanism, EPaxos allows operations without dependencies to be executed concurrently. However, it does not aware the consistency requirement at the application layer. Additionally, during operation processing, extensive communication is incurred among geographically distributed replicas, which is especially expensive in wide area networks. Since usually the same set of replica servers are involved for both consistency and consensus processing, the system performance is expected to be improved greatly if these two processes are combined together.

Based on EPaxos, we design a consensus protocol, named CC-Paxos, which provides distributed context consistency model for applications and integrates the processing of system layer consensus and application layer consistency together. The context creation and detection are fulfilled on the server side. Different clients may share the context ID, which can be assisted by the server, such that clients can propose requests sharing the same context independently without communicating with each other.

The followings are the main contributions of this paper: we *a*) define the distributed context consistency model; *b*) present the design and implementation of distributed context model named CC-Paxos, which also integrates system layer consensus with application layer consistency; and *c*) conduct experiments confirm that CC-Paxos achieves better performance in throughput and latency compared with the traditional

1521-9097/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPADS.2016.61

IEEE computer society

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:39:11 UTC from IEEE Xplore. Restrictions apply.

approaches that address consensus and consistency separately in the system and application layers.

The rest of the paper is organized as follows. We define distributed context consistency and compare it with other consistency models in Section II. We describe our protocol CC-Paxos in Section III. Section IV describes the implementation of CC-Paxos and its comparison (CEPaxos). Section V presents experiment results on the throughput and latency of CC-Paxos with various settings. We review related work in Section VI and conclude in Section VII.

II. Distributed Context Consistency

In this section, we introduce the definitions of distributed context and the distributed context consistency model, then explain the difference with other consistency models.

A. Definition

Before defining distributed context consistency, its abstraction model must be first described. There are two basic operations: `read`(obj) = val and `write`(obj, val). They are equivalent to read and write operations in shared-memory systems, file systems or key-value data stores. Values are stored in and retrieved from *logical replicas*, where each logical replica hosts the entire data space. In real-world systems, a single logical replica, or simply referred as *replica*, could consist of a set of nodes, located in one or multiple data centers.

A *distributed context* is the abstraction of the environment in which operations from a group of clients are issued, correlated, processed and executed. The ordering of operations in a distributed context are defined by the following three rules, denoted by $\longrightarrow$:

1) Intra-client ordering: if γ and λ are two operations from a single client within the same context, then $\gamma \longrightarrow \lambda$ if operation γ happens before λ.
2) Inter-client ordering: if γ and λ are two operations from different clients within the same context, $\gamma \longrightarrow \lambda$ if the client states that λ happens after γ.
3) Transitivity: For operations γ, λ and η, if $\gamma \longrightarrow \lambda$ and $\lambda \longrightarrow \eta$, then $\gamma \longrightarrow \eta$.

These rules establish the ordering among operations issued from different clients in a distributed context. In next section we will discuss the interface for clients to specify the consistency requirements between inter-client operations. Clients may also use one or more contexts to control the scope of their operations for the ordering guarantees they desire.

Distributed context consistency requires that the executing order of all operations are consistent with the order defined by the happens-before or happens-after relationship in a distributed context, $\longrightarrow$. In other word, applications can specify the ordering of their operations explicitly and the storage system will enforce it. Like many other consistency models, distributed context consistency allows concurrent operations. If $\gamma \not\longrightarrow \lambda$ and $\lambda \not\longrightarrow \gamma$, then γ and λ are concurrent. Generally, higher degree of operation concurrency improves system efficiency and performance.

The common way to handle conflicts is to assign an order to conflicted operations by predefined rules and enforce the same ordering on all replicas.

B. Distributed Context Consistency vs. Other Consistency Models

Unlike session consistency, distributed context consistency not only guarantees the execution order of operations in a context of a client, but also the execution order of operations in a context shared by multiple clients on different sites.

A number of consistency models have been defined for distributed systems. To satisfy strict consistency [9], a read operation from a location must return the value of the last write operation to that location. It is almost impractical to implement the strict consistency model in distributed system. Sequential consistency [10] requires that "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program", which ensures at least a global ordering on operations. Between strict consistency and sequential consistency, there is linearizability [11] which can be defined as a sequential consistency with real time constraint by considering a begin time and an end time for each operation.

Causal [12] and causal+ consistency [3] ensure that the system respects the causal dependencies between operations. The main difference between causal consistency and distributed context consistency lies in how to establish the order of operations from different clients. For causal or causal+, if a read operation returns the result of a write, the read operation causally happens before that write. While distributed context consistency allows clients to specify the order of operations, e.g. a read must return a write's value. These two consistency models are not mutually exclusive to each other.

FIFO consistency [13] only requires that all processes see writes from one process in the order they were issued from that process. Eventual consistency [14] guarantees liveness of systems but does not guarantee any consistent returning value from operations before they converge.

The strictness of these models is showed in Figure 1 in decreasing order. Distributed context consistency falls between sequential and FIFO consistency, in parallel with Causal and Causal+.

Linearizability > Sequential **> Causal+ > Causal** / **> Distributed Context** **> FIFO > Eventual**

Fig. 1: The strength of different consistency models.

III. Design

In this paper a distributed context consistency and system consensus combined mechanism is proposed to provide the consistent and reliable data storage service, named CC-Paxos.

One strength of distributed context consistency is to allow clients to specify the order between their operations, therefore the storage system must provide corresponding interfaces.

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:39:11 UTC from IEEE Xplore. Restrictions apply.

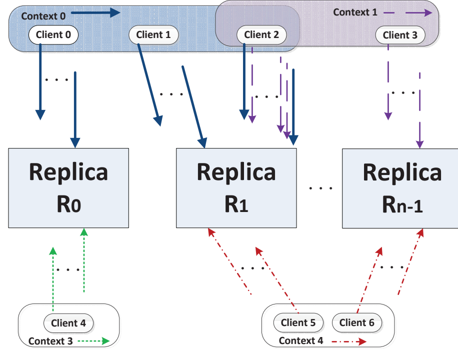

Fig. 2: System architecture.

One way is to assign context-unique sequence numbers to operations in the same context. The system then commits these operations based on their sequence numbers. In this scenario, the applications or the storage system will have to guarantee the uniqueness of sequence numbers. Each operation will be assigned with this context-unique sequence number based on application requirements. In this paper, we propose a timestamp based design in CC-Paxos, which is simple and effective.

CC-Paxos combines the consistency model and the consensus algorithm, thus the terminologies of both are valid. For example, we use requests, commands and operations exchangeably in this paper.

A. System Model

Figure 2 shows the architecture of a distributed storage system across a wide area network using CC-Paxos to guarantee reliability through consensus as well as distributed context consistency. There are n replicas $R = \{R_0, R_1, ..., R_{n-1}\}$, each responsible for handling requests from certain portion of clients in a location based or load based fashion. Clients share contexts, such as $Client_0, Client_1$ and $Client_2$ share $Context_0$, $Client_2$ and $Client_3$ share $Context_1$. A client can be in multiple contexts, e.g., $Client_2$. Meanwhile, a client ($Client_4$) can own a context to itself. The operations from the same context, represented by the same type of lines in the figure, will be sequenced by replicas.

Replicas play different roles for different requests, such as leader, acceptor or learner. Upon receiving a request from a client, replica r becomes the leader for that request. r checks its context ID and timestamp, appoints an unused request processing instance for this request. The instance will be assigned with an ID to represent its order in the overall request processing queue. This instance ID will have to be agreed by all replicas to prevent conflicts. In case one replica has already assigned the same ID to a different request, voting will become necessary. Replica r^*, $r^* \in \{R - r\}$, plays the role of acceptor to update dependency sets and vote for election. The outcome of each used instance is that all the non-fault replicas agree on a single consistent request, which is called that the request is chosen as the value for the instance. The leader replies to the client with the context ID and/or the timestamp if required.

Here, a request being chosen also means that its position in the request sequence is determined, even though some replicas may not necessarily know about it. These replicas are learners of this request. Learner replicas learn the result, and they will commit the request in order, when the operation from the application actually executes. After acknowledgment of committing, the leader will send the confirmation to the client.

B. Client Library and Interface

The CC-Paxos client library provides a simple and s-traightforward programming interface, including creating and destroying contexts, read and write objects within contexts. These are quite similar to existed interfaces except that all functions take a context argument, used by clients to identify the context they are sharing. The storage system uses context to organize operations from geographically distributed clients and uses timestamp for sequencing.

To provide distributed context consistency, the interface has an additional pair of read and write operations, shown as the following:

1) $value \longleftarrow dcc_read(obj, cxt, timestamp)$
2) $bool \longleftarrow dcc_write(obj, value, cxt, timestamp)$

These two APIs allow clients to specify desired timestamps for given operations, which should be in future of the current system time. Otherwise, the operation will return with a failure. An operation with the marked timestamp t should happen after all operations with marked timestamps eariler than t in the context. A reserved word NOW marks the current system time to the operation for further processing. We choose timestamp for inter-client operation ordering for two reasons. First, it is feasible to implement the synchronized global time with limited uncertainty [15]. Second, it is easy for users to understand and use at the application level. There are other means to order inter-client operations, such as giving each operation a sequence number, Such effort requires clients to maintain global unique numbers that introduces extra communication burden and programming complexity.

C. The Basic Algorithm

CC-Paxos exploits timestamp for operation sequencing in distributed contexts and incorporates dependency checking and conflict resolving in the same communication rounds. Operations are interrelated in two ways: *conflict dependency* and *context interference*. Conflicts occur when there are two "simultaneous" operations from different contexts, γ and λ, writing to the same object, which is called the two writes conflict with each other. If two operations are in the same context and $\gamma \longrightarrow \lambda$, γ and λ are contextually interfered or in context interference.

CC-Paxos chooses commands for pre-ordered instances, attaches attributes for each command, and perfects attributes during the process of choosing commands for instances. The attribute consists of the set of commands which the command contextually interfered with (context dependencies) and the set of commands which the command conflicts with (conflicting

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:39:11 UTC from IEEE Xplore. Restrictions apply.

dependencies). CC-Paxos ordering the command according to the two sets.

CC-Paxos consists of two handling phases: the committing phase and the executing phase. During the committing phase, CC-Paxos chooses a request for an instance and computes the dependency set which will be used in the executing phase to order the execution sequence. The two phases and their error handling are described in details as following.

Algorithm 1 Pseudocode of CC-Paxos's commit algorithm

1: **procedure** PHASE 1: ESTABLISH ORDERING CONSTRAINS
2: Replica L on receiving $Request(\gamma, C, ts)$ from a client becomes the designated leader for request γ with predefined time ts in context C:
3: **if** $C == NULL$ **then** $C_\gamma \leftarrow New(Context)$
4: **else** $C_\gamma \leftarrow C$
5: **end if**
6: increment instance number $i_L \leftarrow i_L + 1$
7: $\triangleright$ $CfInterf_{L,\gamma}$ and $CxtDep_{L,\gamma}$ are the sets of instance $Q.j$ such that the request recorded in $cmdsL[Q][j]$ conflict interferes with γ and γ context depends on respectively.
8: **if** $ts < time.NOW$ **then**
9: send error message to client, stop the process
10: **else** $ts_\gamma \leftarrow ts$, wait until ts
11: **end if**
12: send $Precmmit(\gamma, C_\gamma, ts_\gamma)$ to client
13: $seq_\gamma \leftarrow 1 + max(\{cmdsL[Q][j].seq | \; Q.j \in CfInterf_{L,\gamma}\} \bigcup \{0\})$
14: $cfdeps_\gamma \leftarrow CfInterf_{L,\gamma}$
15: $cxtdeps_\gamma \leftarrow CxtDep_{L,\gamma}$
16: $cmdsL[L][i_L] \leftarrow (\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, C_\gamma, ts_\gamma, pre-accepted)$
17: send $PreAccept(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, C_\gamma, ts_\gamma, L.i_L)$ to replicas in $\Gamma \setminus \{L\}$, where Γ is the set of replica in the system

18: **Any replica** R, on receiving
19: $PreAccept(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, C_\gamma, ts_\gamma, L.i_L)$:
20: update $cfdeps_\gamma \leftarrow cfdeps_\gamma \bigcup CfInterf_{R,\gamma}$
21: update $cxtdeps_\gamma \leftarrow cxtdeps_\gamma \bigcup CxtDep_{R,\gamma}$
22: $cmdsR[L][i] \leftarrow (\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, ts_\gamma, pre-accepted)$
23: reply $PreAcceptOK(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, L.i_L)$ to L

24: **Replica** L **(command leader for** γ**),** on receiving $N-1$ $PreAcceptOK$ responses:
25: **if** received $PreAcceptOKs$ from all replicas in $\Gamma \setminus \{L\}$, with $seq_\gamma, cfdeps_\gamma$ and $cxtdeps_\gamma$ the same in all replies **then**
26: run $Commit$ phase for $(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma)$ at $L.i$ (Fast path)
27: **else**
28: update $cfdeps_\gamma \leftarrow Union(cfdeps_\gamma \; from \; all \; replies)$
29: update $seq_\gamma \leftarrow max(\{seq_\gamma \; of \; all \; replies\})$
30: update $cxtdeps_\gamma \leftarrow Union(cxtdeps_\gamma \; from \; all \; replies)$
31: run $Paxos-Accept$ phase for $(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma)$ at $L.i$ (Slow path)
32: **end if**
33: **end procedure**

34: **procedure** PHASE 2: $Paxos-Accept$
35: Command leader L, for $(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma)$ at instance $L.i$:
36: $cmdsL[L][i_L] \leftarrow (\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, accepted)$
37: send $Accept(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, L.i)$ to at least $\lfloor N/2 \rfloor$ other replicas in $\Gamma \setminus \{L\}$
38: Any replica R, on receiving $Accept(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, L.i)$:
39: $cmdsR[L][i] \leftarrow (\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, accepted)$
40: reply $AcceptOK(\gamma, L.i)$ to L
41: Command leader L, on receiving at least $\lfloor N/2 \rfloor$ $AcceptOK$s:
42: run $Commit$ phase for $(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma)$ at $L.i$
43: **end procedure**

44: **procedure** COMMIT
45: Command leader L, for $(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma)$ at instance $L.i$:
46: $cmdsL[L][i_L] \leftarrow (\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, committed)$
47: send commit notification for to client
48: send $Commit(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, L.i)$ to all other replicas in $\Gamma \setminus \{L\}$
49: Any replica R, on receiving $Commit(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, L.i)$:
50: $cmdsR[L][i] \leftarrow (\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma, committed)$
51: **end procedure**

1) The Committing Phase: The algorithm of the committing phase is shown in Algorithm 1. Every replica uses its own command log to record the state of all instances that it has processed. The committing phase consists of Phase 1, Phase 2 and Commit, and not all phases are executed for every command: if the command can be committed through fast path, it can skip Phase 2; if the command can not meet the fast path requirement, it must experience the all phases.

Upon receiving a read/write request for command γ from a client, replica L starts Phase 1 and becomes γ's leader. The replica assigns a new context ID to this operation, i.e. the operation is in its own context and can be executed concurrently with all the others. If command γ specifies a time later than the current time, L will put it into the waiting queue; Error will be returned if the timestamp is earlier than the current time. L replies a $PreCommit$ message with γ's context ID (C_γ)and timestamp (ts_γ) to the client and allocates the next unused instance to the command, attaching the following attributes:

- $cfdeps_\gamma$ (conflict dependencies) is the list of all instances that contains commands conflicting with γ;
- $cxtdeps_\gamma$ (context dependencies) is the list of all instances that contains commands that γ depends on within the same context;
- seq_γ is the sequence number used to break conflict dependency cycles during the execution phase; seq_γ is guaranteed to be larger than the sequence IDs of commands in $cfdeps_\gamma$.

As the leader, L sends command γ and initial attributes in a *PreAccept* message to all replicas R ($R \in \Gamma \setminus \{L\}$, Γ is the set of all replicas in the system). Receiving *PreAccept* message, replicas R updates $cfdeps_\gamma$, $cxtdeps_\gamma$ and seq_γ according to its own information about commands, then replies the new attributes to L. If L receives replies from all replicas R, and all updated attributes are the same, L starts the *Commit* phase (called *fast path*); if the replies are not the same, γ must be committed through slow path. L updates the attributes accordingly: $cfdeps_\gamma$ and $cxtdeps_\gamma$ are the union of counterparts from all replies respectively, seq_γ is set to be the highest seq. Then L starts Phase 2 and tells at least a majority of replicas to accept these attributes. After getting a majority of positive replies, L engages on the *Commit* phase. L changes the status of γ to be committed, and tells all replicas to commit the command by a message containing $cfdeps_\gamma$, $cxtdeps_\gamma$ and seq_γ. All replicas update the attributes and change the status of γ according to the commit message. Command γ committed through slow path can be regarded as running classic Paxos to choose $(\gamma, seq_\gamma, cfdeps_\gamma, cxtdeps_\gamma)$.

2) The Execution Phase: If command γ has been committed in instance $R.i$, in order to execute it, a replica should follow three steps: building the dependency graph, finding the strongly connected components and topological sorting, and executing in the verse topological order.

Building the dependency graph: add γ and all commands in instances from γ's $cfdeps_\gamma$ and $cxtdeps_\gamma$ as nodes, with directed edges from γ to these nodes, repeating this process

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:39:11 UTC from IEEE Xplore. Restrictions apply.

recursively for all of γ's dependencies (both conflict dependencies and context dependencies).

Finding the strongly connected component and topological sorting: several algorithms can compute strongly connected components in linear time, we use Tarjan's algorithm [16] in CC-Paxos to find the strongly connected components, and then sort them topologically.

Executing in the verse topological order: In the verse topological order, for each the strongly connected components:

1 Sort all commands in the strongly connected component by their sequence number, which generates a command sequence called *CDsequence*;

2 For each context cxt_x (which is the context number of at least one command in *CDsequence*), find all commands whose context ID is cxt_x in *CDsequence*, and order them in timestamps increasingly on their previous locations, which does not affect the location of other commands that have different context IDs.

3 Execute every un-executed command according to the order in *CDsequence*, marking

3) Failure Tolerance: After receiving the *PreCommit* reply for the previous request, a client can propose the next. When the command is executed, the client will receive a Commit message.

If a client fails or goes offline after sending a request, the request will be executed and when the client recovers later, it will learn the outcome. If abnormal situation happens during the process of handling a request due to network or some other problems but the replica failure, the system can not commit the request in a normal way, the leader replica will send a commit failure message and ask the client to re-propose or give up the request. The leader replica sets a null operation in the failed instance, marks it as executed and informs other replicas. In this way, CC-Paxos can avoid that an abnormal request blocking the execution of committed requests.

If a replica fails, it can recover according to the log which persists states of all instance they have seen no matter committed or not to disk. Then the failed replica communicates with other normal replicas to learn the committed instances and indicates other replicas to learn instances that have been committed in this replica. For an uncommitted instance, its leader replica will restart the commit phase.

IV. Implementation Consideration

We implement CC-Paxos in Go language and decided to use TCP as the transport protocol in our implementation. For each command originated from a client, we record its context number and timestamp.

A. Recording Context Information

In order to compute context dependency, we use the *map* structure in Go to hold the information of every context, which is stored as a <contextID, instances> pair in the map. The $instances$ is a list of instances which share the same context ID as $contextID$. An instance includes a command, a leader replica, and an instance ID. Since the context ID and timestamp of an instance are the same as those of its command, we use them exchangeably in this paper. The $instances$ in a context is sorted by the timestamp in ascending order. Whenever a replica R_i sees a new instance I with context ID C_I, it puts the information of this instance into the $instances$ associated with C_I in the *map*. The size of the map grows as the replica handles more instances, so cleanup is necessary. The instances with command finished execution can be removed. Since $instances$ is sorted, cleanup based on timestamp is straightforward.

B. Keeping the Dependency Set Small

One challenge in our protocol is that, the dependency sets of arriving request may ($cfdeps$ and $cxtdeps$) grow over time. To efficiently store and transmit $cxtdeps$, we should keep $cxtdeps$ as small as possible.

For each instance, $cxdeps$ can only include the nearest context dependency: the instance number $R.i$ with timestamp ts, $R.i$'s context must be the same with γ, and ts is smaller than ts_γ but is the biggest the system has already known in this context. For an instance, its leader replica sends attributes of the instance to other replica. The other replicas find the instance's nearest context dependency that they have seen, and then reply the nearest context dependency to leader replica in *PreAcceptOK* message. After receiving N-1 *PreAcceptOK* messages, leader replica picks out the instance whose timestamp is the biggest in the *PreAccetOK*s and leader replica's $cxtdeps$, and the chosen instance is the only element in $cxtdeps$ when the instance is committed.

C. The Comparison Target

We compare CC-Paxos against the implementation of causal consistency plus EPaxos (or CEPaxos in short). We choose causal+ consistency [3] and EPaxos [8] as the comparison target, because they offer strong consistency models and consensus protocols. More importantly, they perform the best in practice to the best of our knowledge.

The comparison target, CEPaxos, adopts EPaxos to guarantee system consensus and handle conflicts at the system layer. Clients of CEPaxos achieve causal+ consistency by dispatching requests at the application layer: 1) All clients send requests that have causal dependencies to one client selected as the consistency leader. Other dependency-free requests are sent to the replica servers directly. 2) The consistency leader maintains request queues, each corresponding to a group of causally related requests, which is named as causal group, or *CG* in short. When receiving a request, the leader client puts it into the corresponding queue. 3) Request from different queues are processed concurrently. Requests from the same queue are processed in the order of their causal dependencies. Specifically, one request is processed after successful executions of all its parents. For fair comparison, we also implement CEPaxos in Go language.

V. Evaluation

This section presents the evaluation result of CC-Paxos and CEPaxos.

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:39:11 UTC from IEEE Xplore. Restrictions apply.

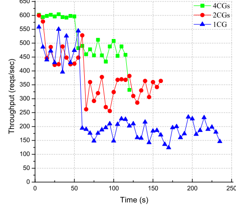

Fig. 3: The throughput of CEPaxos with the variable causal groups number, the self-interference rate 50% and the key number 64.

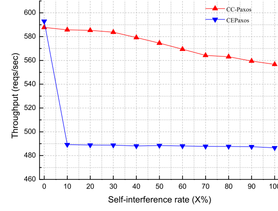

Fig. 4: The throughput of CC-Paxos and CEPaxos with different self-interference rates.

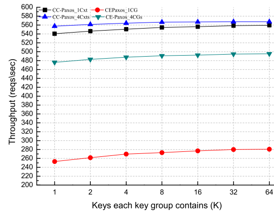

Fig. 5: The throughput of CC-Paxos and CEPaxos with different key group size.

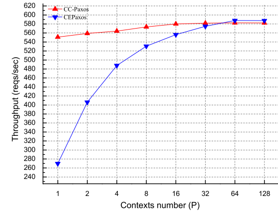

Fig. 6: The throughput of CC-Paxos and CEPaxos with varying number of contexts.

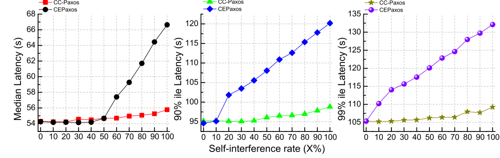

Fig. 7: Median, 90%ile and 99%ile execution latency of CC-Paxos and CEPaxos with different self-interference rates.

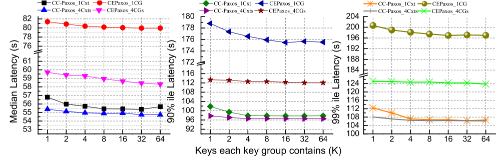

Fig. 8: Median, 90%ile and 99%ile executing latency of CC-Paxos and CEPaxos with various size of key group.

A. *Experimental Setup*

The experiments are conducted on a cluster of machines with Intel Pentium Dual-Core E5400 @2.70 GHz CPU, 16 GB Memory and CentOS 6.2 OS. The network between replicas and clients/replicas is configurable to simulate various deployment scenarios. In practice replicas are usually distributed over WAN, while a client and its local replicas are within the same LAN. LAN is set to have bandwidth of 1,000 Mbps and latency of 0.1 ms, and for WAN these are set to 2 Mbps and 1 ms respectively.

For both CC-Paxos and CEPaxos, clients send read/write requests to replicas, with request containing the following information: 1) an 8-bit operation type with value of read or write; 2) key in 64 bits; and 3) value in 64 bits. The request of CC-Paxos includes two more variables in the command: context ID and timestamp, both in 64 bits.

B. *Workloads*

Clients in CC-Paxos and CEPaxos generate dynamic workload. Clients choose the read or write operation randomly, because in distributed consistency scenarios like CC-Paxos and CEPaxos, reads and writes are typically handled in the same fashion. One of the most important characteristics of workloads are the *interference ratio* and the *conflict ratio* among commands. They are respectively defined as the possibility of two commands belong to the same context and the possibility of two commands targeted at the same key. These two ratios are determined by the number of contexts, the number of keys and the distributions of context and key in commands.

We set up N replicas with M clients, $C = \{C_0, C_1, \cdots, C_{M-1}\}$. A set of P contexts is $Cxt = \{Cxt_0, Cxt_1, \cdots, Cxt_{P-1}\}$, and each context is related to a key group. Consequently there are P key groups, $KG = \{KG_0, KG_1, \cdots, KG_{P-1}\}$. For a key group, K keys are randomly selected out of all keys. A context is shared among T different clients chosen out of C, and this context is also called the clients' designated context. Each client has only one designated context. In CEPaxos, P represents the number of causal groups.

When a client generates a command, its context and key are selected as follows. X is an integer between 0 and 100. For each request, an integer is generated from 0 to 100 at random and if the integer is less or equal than X, the command's context is set to the client's designated context, and its key is picked from the key group randomly. Otherwise, the request is context-free and a random key from the whole key set is assigned to the command. $X\%$ is called as the *self-interference rate* and it can be derived that the interference ratio equals to $\frac{(X\%)^2}{P}$. Meanwhile, the conflict ratio equals to $\frac{1}{P*K}$.

During the experiment, clients send requests in an open loop [17]. Replicas reply to clients only after executing requests. The system throughput is measured at the client side as it receives the reply. The latency is defined as the time between sending the request and receiving the reply from the replica. We also evaluate the median, 90%ile and 99%ile executing latency of all requests for thorough study of the performance. When we evaluate latency, clients send 64000 requests in total. We configure 20 clients per context during the experiments.

Fig. 3 shows the throughput of CEPaxos with 50% self-interference rate. We set the number of context group to 1, 2, and 4, so the interference rate of these context groups are 25%, 12.5% and 6.25% respectively. It can be seen that each trace consists of two phases and the performance of the first phase (0-50 seconds in X-axis) is much higher than the later one. The reason is that when all requests arrive at the beginning, there are limited dependency between these new requests and these causal dependency-free requests are handled quickly concurrently. After these requests are handled, the remaining requests are strictly ordered by their causal dependency within each causal group. The latter phase represents the stable and accurate throughput performance of CEPaxos. This phase is the research focus of this paper.

C. *The Self-interference Rate*

It is expected that the self-interference rate has major effect on the performance of both CC-Paxos and CEPaxos. Higher self-interference rate will result in worse performance

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:39:11 UTC from IEEE Xplore. Restrictions apply.

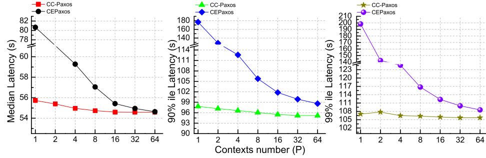

Fig. 9: Median, 90%ile and 99%ile execution latency of CC-Paxos and CEPaxos with varying number of contexts.

in both throughput and latency. To present the effect of self-interference, we set the number of contexts and the size of key group to 4 in this set of experiments.

Fig. 4 and Fig. 7 show the throughput and detailed execution latency of CC-Paxos and CEPaxos with various self-interference rate. The results meet our expectation. In Fig. 4 the throughput of CEPaxos plunges with the self-interference rate grows from zero to 10%. The self-interference rate of 0% means that requests are all dependency-free and can be executed concurrently. In this scenario, the throughput of CC-Paxos is 0.891% less than that of CEPaxos, because CC-Paxos has slightly more overhead on computing request dependency set in committing phase and building request dependency graph in execution phase. For non-zero self-interference rates, the causal constraints on the request order limit the performance of CEPaxos significantly.

In general, the throughput of CC-Paxos significantly outperforms CEPaxos owing to the efficient processing of concurrent requests. CC-Paxos's performance is relatively stable with the increasing self-interference rate.

For the median latency, CC-Paxos starts to outperform CEPaxos at the self-interference rate between 50% to 60%. While for the 90%ile and 99%ile latency, CC-Paxos achieves better performance even with self-interference at 10%. CC-Paxos consistently performs better in latency than CEPaxos.

D. The Size of Key Group

We also examine how the number of keys K in each key group affects system performance. In this set of experiments, we set the number of contexts P to one or four for comparison, and set the self-interference rate X to 70%.

Fig. 5 and Fig. 8 show the influence of K on throughput and latency of CC-Paxos and CEPaxos respectively. It is obvious that with different number of contexts, the throughput and latency performance of both algorithms improves as K increases. Both algorithms perform better with higher number of contexts. This is because with more keys in key groups and more contexts, the targeted keys of the commands become more spread, resulted in decreased confliction rate. Thus the average conflict dependencies among requests decreases, which reduces the times to recursively compute conflict dependency set and building dependency graph, and more importantly, reduces the coordination among clients and replicas for consistency guarantee.

Shown in Fig. 5, the throughput of CC-Paxos consistently performs better. With 4 contexts, the throughput of CC-Paxos is about 17% higher than that of CEPaxos. The performance difference increases to 116% with the number of context of 1. Such performance gain is achieved by the efficiency in detecting concurrently executable requests.

Fig. 8 illustrates the median, 90%ile and 99%ile execution latency of both algorithms. CC-Paxos performs better than CEPaxos regardless the number of keys in key groups and the number of contexts.

E. The Number of Contexts

As shown before, the number of contexts P influences the effect of the interference rate and the conflict rate. In this section we directly measure the effect of P on the system performance. We set the interference rate and the size of key groups to 70% and 64 respectively.

Fig. 6 shows that when the number of contexts is smaller than 32, the throughput of CC-Paxos outperforms CEPaxos by 101% to 204%. The less number of contexts, the higher the advantage in CC-Paxos. In CEPaxos, the causal constraints of the request ordering is enforced on higher number of requests thus dramatically limits its performance. CC-Paxos reduces the number of requests effected by causal constraints, and achieves higher system throughput. With the number of contexts bigger than 64, CEPaxos reaches similar throughput as CC-Paxos. This is because with high number of contexts, the dependency among requests are lowered sufficiently to have no noticeable effect on system throughput.

As presented in Fig. 9, the 90%ile and 99%ile latency of CC-Paxos are much smaller than that of CEPaxos. For median latency, CC-Paxos is much lower than EPaxos until P grows to 32 or higher.

VI. Related Work

The key idea of our work is to determine and guarantee the order of operations required by the logic of application and system. The related work includes three categories.

Consistency models. Most of the early large-scale data stores [1], [2] implement eventual consistency in exchange for low latency. Efforts in [18] provide weak consistency guarantees to increase system performance. However, weak consistency guarantees can be elusive and hard to program to satisfy the complex requirements from applications as pointed out by [15]. Cassandra [19] implements configurable eventual and linearizability consistency. PNUTS [20] provides per-key but not inter-key sequential consistency. COPS [3], a geo-replicated key-value store, tracks dependency between operations to provide causal+ consistency. Operation dependencies are also used to provide read-only/write-only transaction support. COPS improves fault tolerance by exploiting chain replication within a cluster to address node failures. This design, however, is not integrated with the consistency control component, and the performance of causal consistency and chain replication is not evaluated in experiments. C. Lee, et al. [21] aim to provide the strongest consistency, linearizability, through guaranteeing exactly-once semantics. Our proposed distributed session consistency is weaker than linearizability but is sufficient for many applications.

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:39:11 UTC from IEEE Xplore. Restrictions apply.

The definition of *session* is proposed in [5] as an abstraction for the operation sequence during the execution of an application. Whether sessions can be shared across applications, processes or hosts is left unspecified. In this paper we clearly define distributed session consistency and propose CC-Paxos that enforces the defined model.

Consensus protocols. EPaxos [8] achieves high performance under certain conditions with a leaderless approach. With conflict detection mechanism, EPaxos allows commands with no relationship among each other to be executed in any order. However, it does not support strong consistency demanded by the upper layer.

MetaSync [22] is a reliable file synchronization service and implements a variant of Paxos called pPaxos. pPaxos provides consistent updates on top of the unmodified APIs exported by existing services. pPaxos is a client-based Paxos algorithm that demands no direct client-client or server-server communication.

Concurrency control in transactions. Transaction in databases is another area demanding concurrency control. Sinfonia [23] uses two-phase commit (2PC) with optimistic concurrency control [24]. Granola [25] exchanges timestamp between servers to order conflicting transactions. Calvin [26] uses a separate sequencing layer to assign a deterministic locking order to all transactions. A distributed database transaction emphasizes ACID and it is usually issued from a client and fulfilled at multiple servers. The operations in our discussion has no requirements on atomicity and isolation, and may be issued from different clients and should be committed to the quorum of replicas.

Authors in [27] assumes that a transaction consists of a collection of atomic pieces and an atomic operation executes on an independent server. It adopts the similar idea as E-Paxos and CC-Paxos where dependencies between concurrent transactions are tracked and are sent to all servers to resolve conflicts at the commit time.

VII. Conclusion

This paper presents CC-Paxos, the first protocol to satisfy both strong consistency demanded by the application layer and consensus demanded by the system layer. CC-Paxos achieves operation sequencing in distributed contexts by exploiting timestamps and resolves conflicts by dependency checking in the same communication rounds. Our evaluation demonstrates that compared with traditional methods completely isolating the application layer and system layer, CC-Paxos achieves significantly better performance in latency and throughput, with very limited overhead and no sacrifice on scalability. At the same time CC-Paxos also provides simple and elegant APIs for users to specify the inter-client consistency requirements.

References

[1] G. DeCandia, D. Hastorun, and et al., "Dynamo: Amazon's highly available key-value store," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2007, pp. 205–220.

[2] "Voldemort," http://www.project-voldemort.com/.

[3] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. New York, NY: ACM, 2011, pp. 401–416.

[4] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency availability and convergence," University of Texas at Austin, Tech. Rep., May 2011.

[5] D. B. Terry and et al., "Session guarantees for weakly consistent replicated data," in *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, Washington, DC, 1994, pp. 140–149.

[6] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

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

[8] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *ACM Symposium on Operating Systems Principles*, New York, NY, 2013, pp. 358–372.

[9] A. Tanenbaum and M. Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, Inc., 2006.

[10] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 690–691, 1979.

[11] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 3, pp. 463–492, 1990.

[12] M. Ahamad and et al., "Causal memory: definitions, implementation, and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[13] R. J. Lipton and J. S. Sandberg, "Pram: A scalable shared memory," Princeton University, Tech. Rep., 1988.

[14] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[15] J. C. Corbett, J. Dean, and et al., "Spanner: Google's globally distributed database," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 261–264.

[16] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[17] B. C. Kuo, *Automatic Control Systems*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1981.

[18] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *Proceedings of the 32nd International Conference on Very Large Data Bases*, 2006, pp. 715–726.

[19] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Operating System Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[20] B. F. Cooper and et al., "Pnuts: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008.

[21] C. Lee, S. J. Park, A. Kejriwal, S. Matsushitay, and J. Ousterhout, "Implementing linearizability at large scale and low latency," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles*. Monterey, CA: ACM, October 2015, pp. 71–86.

[22] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall, "Metasync: File synchronization across multiple untrusted storage services," in *2015 USENIX Annual Technical Conference (ATC'15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 83–95.

[23] M. K. Aguilera and et al., "Sinfonia: a new paradigm for building scalable distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 159–174, 2007.

[24] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transaction on Database System*, vol. 6, no. 2, pp. 213–226, Jun. 1981.

[25] J. A. Cowling and B. Liskov, "Granola: Low-overhead distributed transaction coordination," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC'12)*, vol. 12, 2012.

[26] A. Thomson and et al., "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 1–12.

[27] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting more concurrency from distributed transactions," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, 2014, pp. 479–494.

Authorized licensed use limited to: Wuhan University. Downloaded on September 18,2022 at 06:39:11 UTC from IEEE Xplore. Restrictions apply.