Noname manuscript No.
(will be inserted by the editor)

Logless One-Phase Commit Made Possible for Highly-Available Datastores

Yuqing Zhu · Philip S. Yu · Guolei Yi · Mengying Guo · Wenlong Ma · Jianxun Liu · Yungang Bao

Received: April 10, 2017 / Revised: April 16, 2018 / Accepted: February 12, 2019

Abstract Highly-available datastores are widely deployed for Internet-based applications. However, many Internet-based applications are not contented with the simple data access interface provided by highly-available datastores. Distributed transaction support is demanded by applications such as massive online payment used by Alipay, Paypal or Baidu Wallet. Current solutions to distributed transaction can spend more than half of the whole transaction processing time in distributed commit. The culprits are the multiple write-ahead logging steps and communication roundtrips in the commit process.

This paper presents the **HACommit** protocol, a logless one-phase commit protocol for highly-available datastores. HACommit has transaction participants vote for a commit before the client decides to commit or abort the transaction; in comparison, the state-of-the-art practice for distributed commit is to have the client decide before participants vote. The change enables the removal of both the participant's

This is a post-peer-review, pre-copyedit version of an article published in the journal of Distributed and Parallel Databases. The final authenticated version is available online at: http://dx.doi.org/10.1007/s10619-019-07261-2

This work was done when J. Liu was with Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

Y. Zhu, M. Guo, W. Ma, Y. Bao
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
E-mail: {zhuyuqing,guomengying,mawenlong,baoyg}@ict.ac.cn

P. S. Yu
University of Illinois at Chicago, Chicago, USA
E-mail: psyu@uic.edu

G. Yi
Baidu, Beijing, China
E-mail: yiguolei@baidu.com

J. Liu
Forcaster Information Tech., Beijing, China
E-mail: liujianxun1106@163.com

write-ahead logging and the coordinator's write-ahead logging steps in the distributed commit process; it also makes possible that, after the client initiates the transaction commit, the transaction data is visible to other transactions within one communication roundtrip time (i.e., one phase). In the evaluation with extensive experiments, HACommit outperforms recent atomic commit solutions for highly-available datastores under different workloads. In the best case, HACommit can commit in one fifth of the time the widely-used two-phase commit (2PC) does.

Keywords
atomic commit; transaction; consensus; replication; two-phase commit; OLTP

1 Introduction

Internet-based services have strong requirements on availability; their data storage widely exploits highly-available datastores [5,42,47]. For highly-available datastores, distributed transaction support is highly desirable. In fact, the problem of supporting transactions in highly-available datastores has attracted interests from both the academia [30, 39, 41] and the industry [15, 19, 34]. Distributed transactions can simplify application development and facilitate massive online transacting business like Paypal [4], Alipay [1] and Baidu Wallet [2]. Besides, it can enable quick responses to big data queries through materialized view and incremental processing [22, 44]. The benefits of transactions come from the ACID (atomicity, consistency, isolation and durability) guarantees [10]. The atomic commit process is key to the guarantee of ACID properties.

The state-of-the-art practice for atomic commit in highly-available datastores is to have the transaction client decide before the transaction participants vote [15, 21, 30, 35, 41, 51], denoted as the *vote-after-decide* approach. Deciding to commit a transaction, the client initiates a distributed commit process, which typically incurs two phases of processing. In the first phase, participants vote on the decision and force write-ahead logs, while the coordinators record the votes through write-ahead logging (WAL) [37]. The second phase is for the coordinator notifying the commit outcome and the participants applying transaction changes. The transaction client can be notified of the commit outcome at the end of the first phase [30, 35, 51], but the commit is not completed and the transaction result is not visible to other transactions until the end of the second phase. The two processing phases involve at least two communication roundtrips, as well as the WAL steps of the coordinator and the participants. The WAL steps and the communication roundtrips are costly procedures in distributed processing.

A different approach to distributed commit is having participants vote for a commit before the client decides to commit or abort the transaction, denoted as the *vote-before-decide* approach. Having the participants vote first, the voting step can overlap with the processing of the last transaction operation, saving one communication roundtrip; and, the votes can be replicated in the course, instead of using a separate processing step. This makes the removal of one processing phase possible. On receiving the client's commit decision, the participants can directly commit the transaction locally; thus, the transaction data can be made visible to other transactions within

one communication roundtrip time, i.e., one phase. The above steps can result in a naive logless one-phase commit (1PC) solution, similar to the classic 1PC protocols but without logging and other impractical assumptions [6].

Implemented in highly-available datastores, the naive logless 1PC solution can enable recovery using replicas instead of logs. In recent years, it has been realized that WAL is not the only way to guarantee atomicity and durability; replication is also a feasible choice [29]. As network access becomes faster than disk access, for example, 10Gbps network bandwidth vs. 500MB/s SSD (solid state disk) write speed, using replication to guarantee the correctness of atomic commit becomes far more efficient than logging to disks. Moreover, replication is already widely exploited to guarantee high availability for large-scale datastores. Thus, it is a natural choice to use replication instead of WAL to guarantee the correctness of atomic commit.

1.1 Problems of the Naive Logless 1PC Solution

Using replication instead of logging, the naive logless 1PC solution looks feasible. In the solution, the coordinator no longer keeps logs and acts only as a message forwarder. Hence, we can designate any server to take the coordinator role. To reduce communication costs, we let the client take the coordinator role. If no failures ever occur, this naive logless 1PC solution can be highly efficient.

However, the naive logless 1PC cannot guarantee the transaction atomicity on client failure because contradicting transaction outcomes (i.e., commit and abort) for the same transaction might coexist in the system. A client can fail in two cases: (1) before sending the outcome to any participant; and, (2) sending the outcome to part of participants. In both cases, the system can automatically designate an agent to end the transaction when it detects the client is not active anymore. The following problematic situations can occur.

Faulty Situation 1. In the first case, the agent can directly notify participants to abort the transaction. It is possible that the client is mistakenly considered as failed because of sudden delayed messages between the client and the system. Then messages with inconsistent outcomes from the client and the agent can later arrive at participants simultaneously.

Faulty Situation 2. For the second case, the agent must contact all participants to learn the decision sent by the client. But only one participant receives the client's decision and the participant fails after sending the client's decision to one replica (denoted as R_a); later, a new participant other than R_a is elected from the failed participant's replicas. Then the agent learns from the new participant and other participants that no decision has been made for the transaction. While the client's original intent for the transaction is unknown, the agent can only ask all participants to abort the transaction. Inconsistency arises as R_a is asked by its participant to abort the transaction, while it has already committed the transaction based on the client's decision that R_a received earlier.

Faulty Situation 3. A more complicated situation can also occur. When the client fails after sending the decision messages, an agent takes over and communicates with all participants, which have not yet received the client's decision. The agent thus decides to abort the transaction. But because of reasons like process scheduling, it

takes the agent a long time before sending the first message. The system mistakenly considers the agent as failed and designates a new agent. At the moment, the client's messages arrive at some participants; thus, the new agent decides to commit the transaction after contacting all participants. Then the two agents send their contradicting decisions to all participants, which will receive different outcomes for the transaction, breaking the atomicity of commit.

1.2 The Main Idea of the Solution

The problems described above arise because of **unpredictable message delays**. The unpredictability of message latency is seldom considered previously for small systems, but it is a common phenomenon in massive systems like highly-available datastores [43]. The potential causes of unpredictability include unexpected network spikes [11] and process scheduling [46]. A system with unpredictable message delays is modeled as *asynchronous system* [16]. The consensus problem [7] is among the most researched problems in asynchronous systems. It is about how to reach a consensus among a set of servers that communicate in messages but with unpredictable message delays, regardless of possible server failures; and, it is required that the servers can only choose the consensus value from a given set of values. The most widely known solution to the consensus problem is the Paxos algorithm [31], which is widely used in real-world systems [12, 13].

To address the problems caused by **unpredictable message delays**, we propose to exploit the Paxos algorithm. But two problems remain:

1. How can the solutions to the consensus problem be used to solve the atomic commit problem, as the two problems are widely known to be incomparable [25]?
2. Even if the first problem can be solved, how can the Paxos algorithm solve the atomic commit problem in one phase, as the classic Paxos generally requires two phases [31] to reach a consensus?

For the **first** problem, we propose to take the vote-before-decide approach. We observe that, with the vote-before-decide approach, the commit process is no longer the classic atomic commit problem; rather, it is turned into a new problem that the client proposes a decision to be accepted by participants, whatever the decision is. In other words, this new problem is to reach a consensus among participants, with the consensus value chosen from a value set containing only one element, i.e., the client's decision. Given that the client will only propose commit when all participants vote YES, *this new commit problem is a consensus problem*; and, this condition can be guaranteed simply via a client-side library.

For the **second** problem, we propose to use the unique client as the initial *proposer* of Paxos and the participants as the *acceptors* and *learners*. Proposers are servers making proposals to be chosen as the consensus value. Acceptors are servers choosing the consensus value. Learners are servers that need to learn the consensus from the acceptors. Paxos can reach a consensus among acceptors in a one-phase process, if the proposer is the initial proposer in a run of Paxos [32]. Through a Paxos run, the client can propose its decision, either commit or abort, to be accepted by participants in one phase.

Table 1 A summary of various distributed atomic commit approaches for highly-available datastores.

	Spanner [15]	**MDCC** [30]	**Replicated Commit** [35]	**TAPIR** [51]	**Helios** [41]	**HACommit**
System Model	Async.	Async.	Async.	Async.	Sync.	Async.
Client Coordinated	NO	YES	YES	YES	NO	YES
Coordinator WAL	YES	NO	YES	NO	YES	NO
Participant WAL	YES	YES	YES	NO	NO	NO
Fault Tolerance	$\lceil \frac{n}{2} \rceil$-1	$\lceil \frac{n}{2} \rceil$-1	$\lceil \frac{n}{2} \rceil$-1	$\lceil \frac{n}{3} \rceil$-1	$\lceil \frac{n}{2} \rceil$-1	$\lceil \frac{n}{2} \rceil$-1

1.3 The Contributions

In this paper, we present the HACommit protocol, a logless one-phase commit protocol for highly-available systems. In the design of HACommit, we propose a new procedure for processing the last transaction operation to enable the vote-before-decide approach. To exploit Paxos, HACommit is designed with a transaction context structure to keep Paxos configuration information for the commit process. HACommit runs the Paxos algorithm once for each transaction commit(abort). Without client failures, HACommit runs the one-phase Paxos; on client failures, it runs the classic Paxos algorithm to reach the same transaction outcome among the participants, which act as *would-be* proposers replacing the failed client. Instead of using logs for participant failure recovery, HACommit has participants replicate their votes and the transaction metadata to their replicas when processing the last transaction operation, making full use of the underlying highly-available datastore. HACommit proposes a recovery process for participants to exploit the replicated votes and metadata.

HACommit can not only commit a transaction within one-phase in a highly-available datastore, as other state-of-the-art commit solutions [30, 35, 51], but also make the transaction changes visible to other transactions within one phase, increasing the transaction concurrency. It can be used along with various concurrency control schemes [6, 10], while some state-of-the-art solutions can only work with optimistic concurrency control schemes [30,51]. HACommit can tolerate both client failures and participant replica failures. In the evaluation with extensive experiments, HACommit can commit in less than a millisecond. In the best case, HACommit can commit in one fifth of the time that the widely-used two-phase commit (2PC) does.

Table 1 compares HACommit to the state-of-the-art transactional solutions for highly-available datastores. Among the six solutions, the recent proposal of Helios assumes a synchronous system model to minimize the commit latency. That is, it requires predictable message delays; but the tail-latency effect [18] will render message delays hardly predictable in large-scale systems. All the other five proposals designed based on the asynchronous system model that is commonly assumed for real-world systems. Except for Spanner and Helios, all the other solutions allow the transaction client to coordinate the commit process in some way, and the client failures will not impair the correctness of database. The client coordination reduces one message

from the commit process, without having the client send its request to a coordinator. Removing WAL further reduces message costs. HACommit and TAPIR are the only two solutions not using WAL. However, TAPIR removes WAL and reduces the commit latency at the price of reducing the fault-tolerance level of the underlying system. TAPIR is the only proposal that can just tolerate $\lceil \frac{n}{3} \rceil - 1$ failures, while all the other proposals tolerate $\lceil \frac{n}{2} \rceil - 1$ failures. That is, TAPIR requires at least five replicas to tolerate a single failed replica. Increasing the replica number increases the difficulty in consistency guarantee; and, it also increases the possibility of conflicts between concurrent transactions, leading to a low concurrency level of TAPIR [39]. HACommit does not have to make such a trade-off because it adopts the vote-before-decide approach and solves the new commit problem by exploiting Paxos and the features of transaction processing.

This paper makes the following contributions to the support of transactions in highly-available datastores:

- We propose a new procedure for processing the last operation of a transaction to enable the vote-before-decide approach for atomic commit in highly-available datastores.
- We design HACommit, a logless one-phase commit protocol for highly-available datastores.
- We implement HACommit for highly-available datastores and make a detailed evaluation that demonstrates the performance benefits of HACommit.

Roadmap. Section 2 overviews the design of HACommit. Section 3 details the last operation processing in HACommit and Section 4 describes the commit process. Section 5 presents the recovery processes on client and participant failures. We report our performance evaluations in Section 6. The paper discusses related work in Section 7 and draws the conclusion in Section 8.

2 Overview of HACommit

HACommit is designed to be used in highly-available datastores, which guarantee high availability of data. Highly-available datastores generally partition data into shards and distribute them to networked servers to achieve high scalability. To guarantee high availability of data, each shard is replicated across a set of servers. Except for data replication and distribution, HACommit does not rely on other properties of the underlying datastore. Clients are front-end application servers or any proxy service acting for applications. Clients can communicate with servers of the highly-available datastores. A transaction is initiated by a client. A transaction participant is a server holding any shard operated by the transaction, while servers holding replicas of a shard are called participant replicas.

The implementation of HACommit involves both client and server sides. On the client side, it provides an atomic commit interface via a client-side library for transaction processing. On the server side, it specifies the processing of the last operation and the normal commit process, as well as the recovery process on client or participant failures.

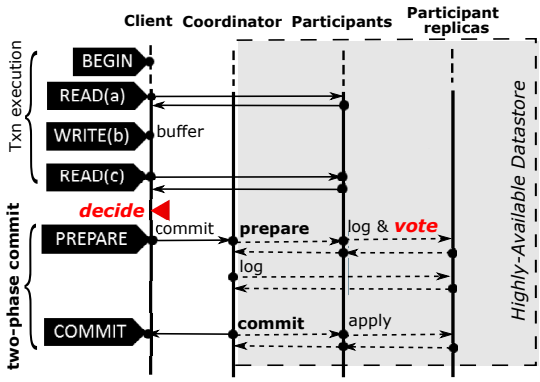

Fig. 1 Transaction processing with two-phase commit in highly-available datastore.

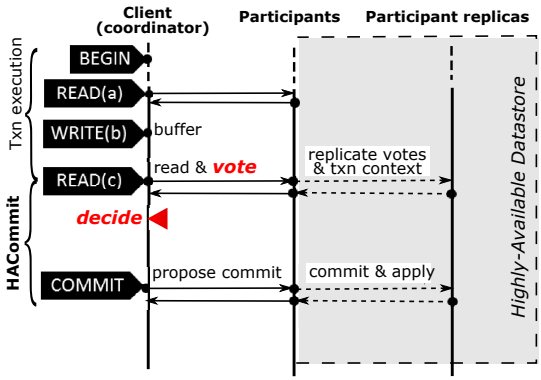

Fig. 2 Transaction processing with HACommit in highly-available datastore, reducing 3 message rounds and 2 logging steps as compared to Figure 1.

A HACommit application begins a transaction, starting from the transaction execution phase. It can then execute reads and writes in the transaction. Except for the last operation, all transaction operations can be processed following either the inconsistent replication solutions [30, 51] or the consistent replication solutions [15, 35]. That is, except for the last operation, HACommit makes no assumptions on other operations of a transaction. On the last operation, the client indicates to all participants that it is the last operation of the transaction. All participants check locally whether to vote YES or NO for a commit, based on the results of local concurrency control, integrity and consistency checks. Different concurrency control schemes (e.g., optimistic, multi-version, and lock-based concurrency control) and isolation levels (e.g., read-committed and serializable isolation levels) [10] can be used with HACommit. Participants replicate their votes and the transaction context information to their replicas respectively before responding to the client. The client will receive votes for a commit from all participants, as well as the processing result for the last operation. This ends the execution phase.

The atomic commit process starts after the client makes its decision to commit or abort the transaction, though the client can only commit the transaction if all partici-

pants vote YES [24]. Then the client proposes its decision on the transaction outcome to the participants and their replicas. The client can safely end the transaction once it has received acknowledgement from a replica quorum of any participant, despite failures of the client or participant replicas. Participant replicas directly commit the transaction on receiving the client's commit decision. An example of transaction processing using HACommit is illustrated in Figure 2.

We also illustrate the transaction processing using two-phase commit for highly-available datastores in Figure 1. 2PC remains the standard protocol for distributed commit in real-world systems. The coordinator role in 2PC cannot be taken by the client as in HACommit because the client can fail and might not come back again. But 2PC relies on a failed coordinator to come back in order to correctly recover the dangling transactions based on the coordinator's log. The 2PC participants vote after the client decides to commit; hence, the transaction *must* be committed, if all participants vote YES for the commit. In comparison, Figure 2 takes the vote-before-decide approach; thus, even if all participants vote YES, the client is still free to commit or abort the transaction. Although HACommit incurs additional communications within the highly-available store when processing the last operation, it still reduces 3 message rounds and 2 logging steps as compared to the 2PC approach. And, HACommit has only one phase in the commit process, as compared to the two phases of 2PC.

3 Processing the Last Operation

On processing the last operation of the transaction, the client sends the last operation to participants holding relevant data, indicating that the operation is the last one. For other participants, the client sends an empty operation as the last operation. All participants process the last operation–those receiving an empty operation do no processing. Thus, the last operation can be a read, a write or an empty operation.

After processing the last operation, participants check locally whether a commit for the transaction can violate any ACID property. Without violation, a participant votes YES; otherwise, it votes NO. If all participants vote YES, the transaction can be correctly committed. The participants replicate their votes and the transaction context to their replicas respectively before responding to the client. The replication of participant votes and the transaction context is required to keep the votes available and guarantee voting consistency in case of participant failures. The participants piggyback their votes on their response to the client's last operation request after the replication. The client makes its decision on receiving responses from all participants.

Transaction context. The transaction context must include the *transaction ID* and the *shard IDs*. The transaction ID uniquely identifies the transaction and distinguishes the Paxos instance for the commit process. The shard IDs are necessary to compute the set of participant IDs, which constitute the configuration information of the Paxos instance for commit. This configuration information must be known to all Paxos acceptors, i.e., transaction participants.

In case when inconsistent replication [51] is used in operation processing, the transaction context must also include *relevant writes*. Relevant writes are writes operating on data held by a participant and its replicas. The relevant writes are necessary

in case of participant failures. With inconsistent replication, participant replicas might not process the same writes for a transaction as the participant. Consider when a set of relevant writes are known to the participant but not its replicas. The client might fail after sending the Commit decision to participants. In the meantime, a participant fails and one of its replica acts as the new participant. Then, the recovery proposers propose the same Commit decision. In such case, the new participant will not know what writes to apply when committing the transaction. To reduce the data kept in the transaction context, the relevant writes can be recorded as commands [36].

4 The Commit Process

In HACommit, the client commits or aborts a transaction by initiating a Paxos instance. The commit process without failures is depicted in Figure 2. In the following, we first describe the background knowledge of Paxos (§ 4.1). Then, we present the Paxos-based one-phase commit process of HACommit. As compared to the common exploitation of Paxos, HACommit makes a few adaptations, including how participants make acknowledgement (§ 4.3), how concurrent transaction commit instances be distinguished (§ 4.4) and how the instance configuration information is kept (§ 4.5).

4.1 Background: the Paxos Algorithm

A run of the Paxos algorithm is called an instance. A Paxos instance reaches a single consensus among the participants, which is called *acceptors* in Paxos. An instance proceeds in rounds. Each round has a ballot with a unique number *bid*. Any would-be proposer (i.e., potential future proposers) can start a new round on any (apparent) failure. Each round generally consists of two phases [13] (*phase-1* and *phase-2*), and each phase involves one communication roundtrip. The consensus is reached when one active proposer successfully finishes one round. Then, the learners will learn from the acceptors the reached consensus. In an instance of Paxos, if a proposer is the *only* and the *initial* proposer, it can propose any value to be accepted by acceptors as the consensus, incurring one communication roundtrip between the proposer and the acceptors [32].

Paxos is commonly used in reaching the consensus among a set of replicas. Each Paxos instance has a configuration, which includes the set of acceptors and learners. Widely used in reaching replica consensus, Paxos is generally used with its configuration staying the same across instances [8]. The configuration information must be known to all proposers, acceptors and learners. In data replication, the set of data replicas are acceptors and learners. The leader replica is the initial proposer and all other replicas are would-be proposers. Clients send their writing requests to the leader replica, which picks one write or a write sequence as its proposal. Then the leader replica starts a Paxos instance to propose its proposal to the acceptors. In practice, the configuration can stay the same across different Paxos instances, e.g., writes to the same data at different time.

Algorithm 1: A client running a transaction

```
   // the transaction tid has n operations
 1 begin the transaction tid;
 2 foreach n − 1 operations do
 3 |   send the operation to hosting participants and collect processing results;
 4 end
   // process the last operation
 5 send the n-th operation to hosting participants;
 6 send cached writes to and inquire votes from all participants;
 7 wait for responses from all participants;
   // the commit begins
 8 foreach participants do
 9 |   set outcome = Commit; bid = 0;
10 |   send a phase-2 message with outcome and the ballot number bid;
11 end
12 wait for responses from all participants or time timeout;
13 if not receiving phase-2 responses from all participants then
   |   // call Algorithm 3
14 end
15 end the transaction tid safely according to responses;
```

4.2 The One-Phase Commit Process

In HACommit, the client is the *only* and the *initial* proposer of the Paxos instance, as each transaction has a unique client. As a result, the client can commit the transaction in one communication roundtrip to the participants.

The commit process starts from the second phase (phase-2) of the Paxos algorithm. That is, the client first sends a phase-2 message to all participants. To guarantee the correctness, the exploitation of the Paxos algorithm must strictly comply with the algorithm specification. Complying with the Paxos algorithm, the phase-2 message includes a ballot number *bid*, which is equal to zero, and the proposal value, which can be *commit* or *abort*. On receiving the phase-2 message, a participant records the ballot number and the outcome for the transaction locally. Then it commits the transaction by applying the writes and releasing all data items; or, it aborts the transaction by rolling back the transaction and releasing all data items. In the meantime, the participant invokes the replication layer to replicate the result to its replicas. Afterwards, each participant acknowledges the client. Alternatively, the client can send the phase-2 message to all participants and their replicas. Each participant replica also follows the same processing procedure as its participant's. Then, the client waits responses from all participants and their replicas. With regard to the above process, Algorithm 1 presents the pseudocodes for the client part and Algorithm 2 presents those for the participant part.

4.3 Participant Acknowledgements

For any participant, if the acknowledgements by a quorum of its replicas are received by the client, the client can safely end the transaction. In fact, the commit process is

Algorithm 2: A participant processing commit messages

```
   // initially, BS_tid ← ∅ for the transaction tid
 1 receive msg_tid from a proposer;
 2 switch msg_tid do
 3     case phase-1
 4         extract bid from phase-1;
 5         if bid > b,∀b ∈ BS_tid then
 6             BS_tid ← BS_tid ∪ {bid};
 7             if ever accepted outcome with bid_a then
 8                 include outcome and bid_a in msg_r;
 9             end
10             notify replicas of msg_r;
11             send msg_r to the proposer;
12         end
13     case phase-2
14         extract bid from phase-2;
15         if bid ∈ BS_tid and bid ≥ b,∀b ∈ BS_tid then
16             accept outcome with bid and record the acceptance;
17             notify replicas of outcome with bid;
18             end the transaction according to outcome;
19             acknowledge the proposer;
20         end
21
22 endsw
```

not finished until all participants acknowledge the client. But any participant failing to acknowledge can go through the failure recovery process (Section 5) to successfully finish the commit process. In HACommit, all participants must finally acknowledge the acceptance of the client's proposal so that the transaction is committed at all data operated by the transaction.

The requirement for participants' acknowledgements is different from that for the quorum acceptance in the original Paxos algorithm. In Paxos, the consensus is reached if a proposal is accepted by more than a quorum of participants. The original Paxos algorithm can tolerate the failures of both participants (acceptors) and proposers. HACommit uses the client as the initial proposer and the participants as acceptors and would-be proposers when exploiting Paxos for the commit process. In its Paxos exploitation, HACommit only tolerates the failures of the initial proposer and would-be proposers. However, the failure of participants (i.e., acceptors) can be tolerated by the participant replication, which can also exploit consensus algorithms like Paxos.

4.4 Distinguishing Concurrent Commits

Each Paxos instance corresponds to the commit of one transaction, but one participant can engage in multiple Paxos instances for commit, as the participant can involve in multiple concurrent transactions. To distinguish different transactions, we include a transaction ID in the phase-2 message, as well as in all messages sent between clients

and participants. A transaction T is uniquely identified in the system by its ID *tid*, which can be generated using distributed methods, e.g., UUID [33].

4.5 Paxos Configuration Information

Different from those Paxos exploitations where the configuration stays the same across multiple instances, HACommit has different configurations in Paxos instances for different transaction commits. The set of participants is the configuration of a Paxos instance. Each transaction has different participants, leading to different configurations of Paxos instances for commit. As required by the algorithm, the configuration must be known to all proposers and within the configuration. A replacing proposer (i.e., a recovery node) needs the configuration information to continue the algorithm after the failure of a previous proposer. The first proposer of the commit instance is the transaction client, which is the only node with complete information of the configuration. If the client fails, the configuration information might get lost. In fact, a client might fail before the transaction comes to the commit step. Then a replacing proposer will hardly have enough configuration information to abort the dangling transaction.

To guarantee the availability of the configuration information, we include the configuration information in the phase-2 message. Besides, as the configuration is expanding and updating after a new operation is processed, the client must send the up-to-date configuration to *all* participants contacted so far on processing each operation. In case that a participant fails and one of its replicas takes its place, the configuration must be updated and sent to all replicas of all participants. The exact configuration of the Paxos instance for commit will be formed right on the processing of the last transactional operation. In this way, each participant replica keeps locally an up-to-date copy of the configuration information. As a participant can fail and be replaced by its replicas, HACommit does not rely on participant IDs for the configuration reference. Instead, it records the IDs of all shards operated by the transaction. With the set of shard IDs, any server in the system shall find out the contemporary set of participants easily.

5 Failure Recovery

In the design of HACommit, we assume the crash-stop failure behavior of server as many real-world implementations of Paxos do. That is, if a client or a participant replica fails, it will fail and stop responding. Servers will not send random and malicious messages on failure. The one-phase commit protocol HACommit will require more rounds of communication on failures. In the following, we describe the recovery mechanisms for client failure and participant replica failure respectively.

5.1 On Client Failure

In HACommit, all participants are all candidates of recovering nodes for a failure. We call recovering nodes as *recovery proposers*, which act as would-be proposers of

Algorithm 3: A proposer resuming the commit process

```
   // initially, BS_tid ← ∅ for the transaction tid
 1 while not receiving phase-2 responses from all participants do
 2     while not receiving phase-1 responses from all participants do
 3         set bid such that bid > b, ∀b ∈ BS_tid;
 4         BS_tid ← BS_tid ∪ {bid};
 5         foreach participants do
 6             send a phase-1 message with bid;
 7         end
 8         wait for responses from all participants or time timeout;
 9     end
10     if receive phase-1 responses from all participants then
11         extract bid_r from all phase-1 responses and add to BS_tid;
12         set outcome to outcome_c that has the highest ballot number in phase-1 responses;
13         if outcome is NULL then
14             set outcome to ABORT;
15         end
16         foreach participants do
17             send a phase-2 message with outcome and the ballot number bid;
18         end
19         wait for phase-2 responses from all participants or time timeout;
20     end
21 end
22 end the transaction tid safely;
```

the commit process. The recovery proposers will be activated on client failure. In an asynchronous system, there is no way to be sure about whether a client actually fails. In practical implementations, a participant can keep a timer on the duration since it has received a message from the current proposer. If the duration has exceeded a threshold, the participant considers the current proposer as failed. Then it considers itself as the recovery proposer.

A recovery proposer must run the complete Paxos algorithm to reach the consensus *safely* among the participants. Multiple rounds, phases and communications roundtrips can be involved on client failures, as any would-be proposer can start a new round on any (apparent) failure.

Although complicated situations can happen, the participants of a transaction will reach the same outcome eventually, if they ever reach a consensus and the transaction ends. For example, as delayed messages cannot be distinguished from failures in an asynchronous system, the current proposer might in fact have not failed. Instead, its last message has not reached a participant, which considers the proposer as failed. Or, multiple participants consider the current proposer as failed and start a new round of Paxos simultaneously. All these situations will not impair the safety of the Paxos algorithm [31].

5.1.1 The Recovery Process

A recovery proposer starts the recovery process by starting a new round of the Paxos instance from the first phase. In the first phase, the new proposer will update the ballot number *bid* to be larger than any one that it has seen. It sends a phase-1 message with

the new ballot number to all participants. On receiving the phase-1 message with *bid*, if a participant has never received any phase-1 message with ballot number greater than *bid*, it responds to the proposer. The response includes the accepted transaction decision and the ballot number on which the acceptance is made, if the participant has ever accepted any transaction decision.

If the proposer has received responses to its phase-1 message from all participants, it sends a phase-2 message to all participants. The phase-2 message has the same ballot number as the proposer's last phase-1 message. Besides, the transaction outcome with the highest ballot number in the responses is proposed as the final transaction outcome; or, if no accepted transaction outcome is included in responses to the phase-1 message, the proposer must propose abort to comply with the transaction definition. Unless the participant has already responded to a phase-1 message having a ballot number greater than *bid*, a participant accepts the transaction outcome and ends the transaction after receiving the phase-2 message. The participant acknowledges the proposer accordingly. After receiving acknowledgements from all participants, the new proposer can safely end the transaction. Algorithm 3 presents the pseudocodes for the above description.

5.1.2 Liveness

Similar to Paxos, HACommit assumes for the guarantee of liveness that one proposer will finally succeed in finishing one round of the algorithm. In HACommit, if all participants consider the current proposer as failed and starts a new round of Paxos simultaneously, a racing condition among new proposers could be formed in the first phase of Paxos. No proposer might be able to succeed in finishing the second phase of Paxos, making the liveness of commit not guaranteed. Though rarely happening, the racing condition among would-be proposers must be avoided in Paxos [31] for the liveness consideration. In actual implementations, the random back-off of candidates, i.e., having the candidates wait for a random time before initiating another rounds, is enough to resolve the racing situation [8, 13], although leader election [13] or failure detection [45] services outside the algorithm implementation can also be used.

5.2 On Participant Replica Failures

HACommit can tolerate not only client failures, but also participant replica failures. It can guarantee continuous data availability if more than a quorum of replicas are accessible for each participant in a transaction. In case that quorum replica availability cannot be guaranteed, HACommit can be blocked but the correctness of atomic commit is guaranteed anyhow [31]. The high availability of data enables a recovery process based on *replicas* instead of logging, though logging and other mechanisms like checkpointing [29] and asynchronous logging [36] can fasten the recovery process.

Failed participant replicas can be recovered by copying data from the correct replicas of the same participant. Or, recovery techniques used in consensus and replication services [12, 17] can be employed for the replica recovery of participants.

Although one replica is selected as the leader (i.e., the participant), the leader replica can easily be replaced by other replicas of the same participant [12]. If a participant failed before sending its vote to its replicas, the new leader will make a new decision for the vote. Otherwise, as the vote of a participant is replicated before sending to the coordinator, this vote can be kept consistent during the change of leaders. Besides, the client has sent the transaction outcome to all participants and their replicas in the commit process. Thus, failed participant replicas can be recovered correctly as long as the number of failed replicas for a participant is tolerable by the consensus algorithm in use.

Generally, there are fewer failed replicas for each participant than that is tolerable by the highly-available datastore, as the number of replicas can be increased to tolerate more failures. Hence, transactions can be committed in HACommit. In case there are not enough active replicas, the participant without enough replicas will not respond to the client so as to guarantee replica consistency and correctness. The commit process will have to be paused until all participants are equipped with enough active replicas. Though the liveness of the protocol relies on the number of active replicas, HACommit can guarantee the correctness of commit and the consistency of data in all situations.

6 Evaluation

Our evaluation explores three aspects of HACommit:

1. **Commit performance** — HACommit has smaller commit latency than other protocols and this advantage increases as the number of participants per transaction increases.
2. **Fault tolerance** — HACommit can tolerate client failures, as well as server failures.
3. **Transaction processing performance** — HACommit has higher throughputs and lower average latencies than other protocols.

6.1 Experimental Setup

We compare HACommit with two-phase commit (2PC), replicated commit (RCommit) [35] and MDCC [30]. Two-phase commit (2PC) is still considered the standard protocol for committing distributed transactions. It considers no replication. RCommit and MDCC are state-of-the-art commit protocols for distributed transactions over replicated data as HACommit is. It has better performance than the approach that layers 2PC over the Paxos algorithm [8, 15]. MDCC guarantee only isolation levels weaker than serializability. In the evaluation, the same concurrency control scheme and the same storage management component are used for HACommit, 2PC and RCommit. These three implementations guarantee the serializability consistency level. Compared to the implementations for 2PC and RCommit, the HACommit implementation also supports the weak consistency level of read committed [9]. The evaluation of MDCC is based on its open source [3].

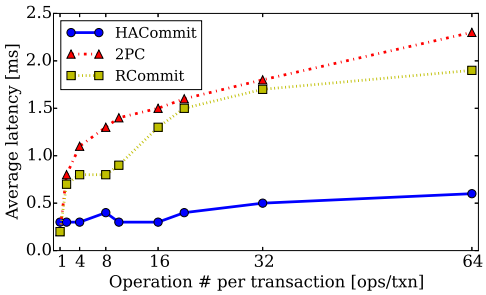

Fig. 3 Commit latencies when increasing the number of operations per transaction.

We evaluate all implementations using the Amazon EC2 cloud. We evaluate each implementation by extending the YCSB benchmark [14] to support transaction processing performance evaluations. As our database completely resides in memory and the network communication plays an important role, we deploy the systems over memory-optimized instances of r3.2xlarge (with 8 cores, 60GB memory and high-speed network). Unless noted otherwise, all implementations are deployed over eight nodes. The cross-node communication roundtrip is about 0.1 milliseconds. Though our evaluation is not carried out based on the fastest network and the fastest durable storage in the world, it suffices to show that the replication-based HACommit protocol is much more efficient than the WAL-based commit protocols as long as the network speed is fast enough.

For HACommit, RCommit and MDCC, the database is deployed with three replicas. For 2PC, no replication is used. Generally, 2PC requires buffer management for durability. We do not include one for 2PC and in-memory database is used instead. The durability is guaranteed through operation logging. As buffer management takes up about one fifth of the local processing time of transactions, our 2PC implementation without buffer management should perform faster than a typical 2PC implementation.

In all experiments, each server runs a server program and a test client program. By default, each client runs with 10 threads. Each data point in the graphs represents the median of at least five trials. Each trial is run for over 120s with the first and last quarter of each trial elided to avoid start up and cool down artifacts. For all experimental runs, clients recorded throughput and response times. We report the average of three sixty-second trials.

For all experiments, we preload a database containing a single table with 10 million records. Each record has a single primary key column and one additional column that is filled with 10 bytes of randomly generated string data. We use small-size records to focus our attention on the key performance factors. Accesses to records are uniformly distributed over the whole database. In all workloads, transactions are committed if no conflicts exist. That is, all transaction aborts in the experiments are due to concurrency control requirements.

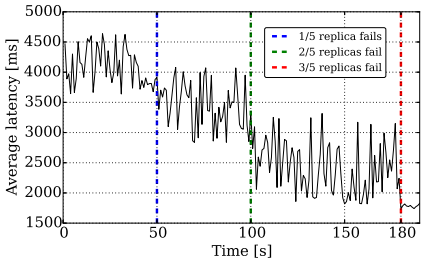

Fig. 4 Transaction latency variations during server failures.

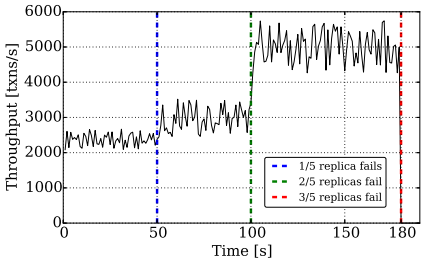

Fig. 5 Transaction throughput variations during server failures.

6.2 Commit Performance

As we are targeting at transaction commit protocols, we first examine the actual costs of the commit process. We study the duration of the commit process. We do not compare the commit process of HACommit with that of MDCC because the latter integrates a concurrency control process; comparing only the commit process of the two protocols is unfair to MDCC.

HACommit outperforms 2PC and RCommit in varied workloads. Figure 3 shows the latencies of commit. We vary the number of operations per transaction from 1 to 64. The advantage of HACommit increases as the number of operations per transaction increases. When a transaction has 64 operations, HACommit can commit in one fifth of the time 2PC does. This performance is more significant as it seems, as HACommit uses replication and 2PC does not. That means, HACommit has $n-1$ times more participants than 2PC in the commit, where n is the number of replicas. The performance advantage of HACommit is due to the reduction of message rounds and logging steps.

HACommit's commit latency increases slightly as the number of operations increases to 20. On committing a transaction, the system must apply all writes and release all locks. When the number of operations is small, applying writes and releasing locks in the in-memory database cost a small amount of time, as compared to the network communication roundtrip time (RTT). As the number of operations increases, the time needed increases slightly for applying all writes in the in-memory database. Accordingly, the commit latency of HACommit increases.

The commit latencies of 2PC and RCommit are increasing as the number of operations per transaction increases. The two protocols need to log new writes for commit and the old values of data items for rollback, thus the time needed for the prepare phase increases as the number of writes goes up, leading to a longer commit process. 2PC has higher commit latency than RCommit, because in our implementations, 2PC must log in-memory data and RCommit relies on replication for fault tolerance.

6.3 Fault Tolerance

In the fault-tolerance tests, we examine how HACommit behaves under client failures and server failures. The evaluation result demonstrates that no transaction is blocked

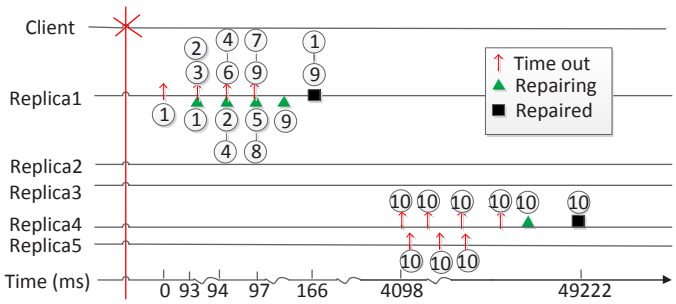

Fig. 6 Logged events showing HACommit's behavior on a client failure (circled numbers are transactions).

under server failures and the client failure, as long as a quorum of participant replicas are accessible.

We use five replicas and initiate one client in the fault tolerance tests. To create failure scenarios, we actively kill a process in the experiments. The network module of our implementations can instantly return an error in such case. Our implementation processes the error as if connection timeout on node failures happens.

The throughput and latency numbers in Figure 4 and 5 indicate that the system is operating normally and the failures can be properly handled; they are not for the performance evaluation purpose. In practice, failures are not frequent and failed replicas will be detected and soon be replaced, with the system returning to the normal state quickly.

Figure 4 shows the evolution of the average transaction latency in a five replica setup that experiences the failure of one replica at 50, 100 and 180 seconds respectively. The corresponding throughputs are shown in Figure 5. The latencies and throughputs are captured for every second. At 50 and 100 seconds, the average transaction latency decreases and the throughput increases. With pessimistic concurrency control, reads in the HACommit implementation take up a great portion of time. The failure of one replica means that the system can process fewer reads. Hence, this leads to lower average latencies and higher throughputs for read transactions, as well as for all transactions. At 180 seconds, we failed one more replicas, violating the quorum availability assumption of HACommit. The throughput drops to zero immediately because no operation or commit process can succeed at all. The HACommit implementation uses timeouts to detect failures and quorum reads/writes. As long as a quorum of replicas are available for every data item, HACommit can process transactions normally.

We also examine how HACommit behaves under transaction client failures. We have all servers log the events of timing out, repairing a transaction and getting a transaction repaired. We deliberately kill the client in an experiment. Each server program periodically checks its local transaction contexts to see if any last contact time exceeds a timeout period. We set the timeout period to be 15 seconds. We have synchronized the clocks of nodes to coordinate records in their logs.

Figure 6 plots the logged events and demonstrates how participant replicas recover from the client failure. In Figure 6, the time axis at the bottom stretches from

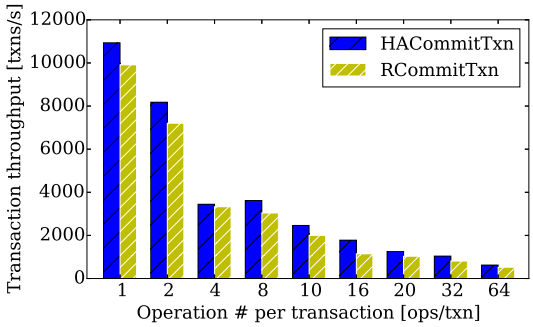

Fig. 7 Transaction throughput: HACommit vs. RCommit.

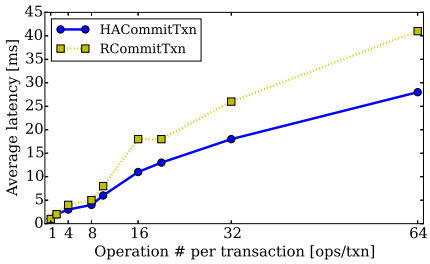

Fig. 8 Average transaction latency: HACommit vs. RCommit.

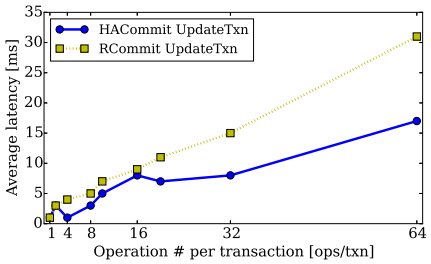

Fig. 9 Latency of update transactions: HACommit vs. RCommit.

left to right. The client fails at the beginning, represented by the cross at the *client* line. The circled numbers represent transactions. Transactions 1 to 9 are detected by replica 1 for not being ended, because there are timeouts on the last time when a processing message is received at replica 1 for these transactions respectively. Timeouts are specified by arrows. Afterwards, replica 1 wins in the commit instances for transactions 1 to 9 and thus initiates repairing processes for the nine transactions respectively. The initiation times and the finished times for the repairing processes are specified by triangles and squares respectively. Replica 1 aborts the nine transactions in the repairing process because no transaction outcome has ever been accepted by any replica.

Transaction 10 is detected for not being ended by replicas 4 and 5 later–this transaction is ended at the other replicas. In our implementation, we deliberately makes replica i wait for i timeout periods to avoid contention. Replica 4 starts a repairing process after four timeouts, when it finds no replicas with smaller IDs promise to repair the transaction. Replica 4 finally commits the transaction. Replica 5 has three timeouts for transaction 10 because the repairing process for the transaction begins before the fourth timeout occurs. And, by that time, the transaction is already committed in the repairing process initiated by replica 4.

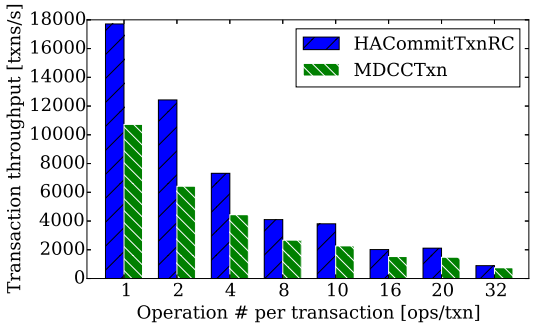

Fig. 10 Transaction throughput under read-committed CC: HACommit vs. MDCC.

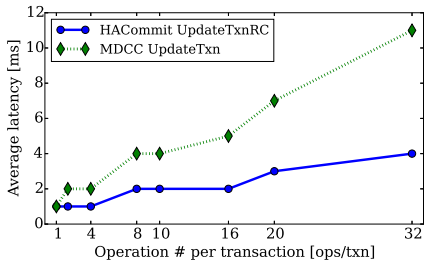

Fig. 11 Latency of UPDATE transactions under read-committed CC: HACommit vs. MDCC.

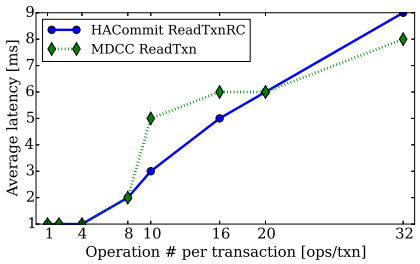

Fig. 12 Latency of READ transactions under read-committed CC: HACommit vs. MDCC.

6.4 Transaction Throughput and Latency

We evaluate the transaction throughput and latency when using different commit protocols. In the experiments, we retry the same transaction on the failure of lock acquisition, until the transaction is successfully committed. Each retry is made after a random amount of time. Figure 7 shows the transaction throughputs when using HACommit and RCommit. The HACommit implementation has larger transaction throughputs than the RCommit implementation in all workloads. The retry policy of the experiments leads to the close proximity between the throughputs of HACommit and RCommit.

Figure 8 demonstrates the average transaction latencies. HACommit has lower transaction latencies than RCommit in all workloads. HACommit's advantage on transaction latency increases as the number of operations in a transaction increases in the workloads. As both implementations use the same concurrency control and isolation level, factors leading to HACommit's advantage over RCommit are two-fold. First, no costly logging is involved during the commit. Second, no persistence of data is needed.

We compare the update transaction latencies of HACommit and RCommit in Figure 9. Both implementations use the same concurrency control scheme and consistency level. We can see that HACommit outperforms RCommit. As the number of operations increases in the workloads, HACommit's advantage also increases. The

advantage of HACommit is still due to a commit without logging and data persistence.

We also examine the transaction throughput and latency when using weaker isolation levels with HACommit. In this case, we compare HACommit against MDCC. We implemented HACommit-RC with the read-committed isolation level [9]. This is an isolation level comparable to that guaranteed by MDCC. HACommit-RC differs from HACommit in that it acquires no locks on reads. Figure 10 shows the transaction throughputs for HACommit-RC and MDCC. The latencies of update transactions and read transactions are shown in Figure 11 and Figure 12.

HACommit-RC has larger transaction throughputs than MDCC in all workloads. The latencies of update transactions are lower in the HACommit-RC implementation than in the MDCC implementation. The reason that HACommit-RC has better performance in transaction throughput and update transaction latency is as follows. MDCC uses optimistic concurrency control, which can cause high abort rates under high contention, leading to lower performance than HACommit-RC, which uses pessimistic concurrency control. Besides, MDCC holds data by outstanding options, leading to the same effect of locking in committed transactions.

HACommit-RC and MDCC have similar performances in read transactions. Both HACommit-RC and MDCC implement read transactions similarly and guarantee the read-committed consistency level. MDCC performs slightly better than HACommit for read-only transactions with reads fewer than 20 due to the optimistic concurrency control scheme. But the performance of the MDCC implementation drops when each transaction has more than 20 operations. After going through the open-source codes of MDCC, we believe it is the implementation of MDCC that makes it perform worse than HACommit when the number of read operations exceeds 20 for a transaction.

7 Related Work

Atomic commit protocols (ACPs). A large body of work studied the atomic commit problem in distributed environment both in database community [6, 23, 37] and distributed computing community [26, 27]. The most widely used atomic commit protocol is two-phase commit (2PC) [10]. It has been proposed decades ago, but remains widely exploited in recent years [15, 29, 35, 38, 49]. 2PC involves at least two communication round trips between the transaction coordinator and the participants. Relying on both coordinator logs and participant logs for fault tolerance, it is blocking on coordinator failures.

Non-blocking atomic commit protocols were proposed to avoid the blocking on coordinator failures during commit. But some assume the impractical model of synchronous communication and incur high costs, so they are rarely implemented in real systems [48]. Those assuming the asynchronous system model generally exploit coordinator replication and the fault-tolerant consensus protocol [23, 26]. These non-blocking ACPs generally incur an even higher cost than 2PC. Besides, they are all designed taking the same vote-after-decide approach as 2PC, i.e., that participants vote after the client decides.

One-phase commit (1PC) protocols were proposed to reduce the communication costs of 2PC. Compared to 2PC, they reduce both the number of forced log writes and communication roundtrips. The price is to send all participants' logs to the coordinator [50] or to make impractical assumptions on systems, e.g, consistency checking on each update [6]. Non-blocking 1PC protocols also exist. They have the same problems as blocking 1PC protocols. Though 1PC protocols have participants vote for commit before the client decides as HACommit does, they do not allow the client to abort the transaction if all transaction operations are successfully executed [10]. In comparison, HACommit gives the client all the freedom to abort a transaction.

All the above atomic protocols do not consider the high availability of data as a condition, thus involving unnecessary logging steps for failure recovery at the participants or the coordinator. Exploiting the high availability of data, the participant logging step can be easily implemented as a process of data replication, which is executed for each operation in highly-available datastores–no matter the operation belongs to a transaction or not.

ACPs for highly-available datastores. In recent years, quite a number of solutions are proposed for atomic commit in highly-available datastores. Spanner [15] layers two phase locking and 2PC over the non-blocking replica synchronization protocol of Paxos [31]. Spanner is non-blocking due to the replication of the coordinator's and participants' logs by Paxos, but it incurs a high cost in commit. Message futures [40] proposes a transaction manager that utilizes a replication log to check transaction conflicts and exchange transactions information across datacenters. The concurrency server for conflict checking is the bottleneck for scalability and performance. Besides, the assumption of shared logs are impractical in real systems [6]. Helios [41] can guarantee the minimum transaction conflict detection time across datacenters. However, it relies on a conflict detection protocol for optimistic concurrency control using replicated logs, which makes strong assumptions on one replica knowing all transactions of any other replica within a critical time interval, which is impossible for asynchronous systems with disorderly messages [20]. The safety property of Helios in guaranteeing serializability can be threatened by the fluctuation of cross-DC communication latencies. These commit proposals heavily exploit transaction logs, while logging is costly for transaction processing [28].

MDCC [30] proposes a commit protocol based on Paxos variants for optimistic concurrency control [10]. MDCC exploits the application server as the proposer in Paxos, while the application server is in fact the transaction client. Though its application server can find out the transaction outcome within one processing phase, the commit process of MDCC is inherently two-phase, i.e., a voting phase followed by a decision-sending phase, and no concurrent accesses are permitted over outstanding options during the commit process. TAPIR [51] has a Paxos-based commit process similar to that of MDCC, but TAPIR can be used with pessimistic concurrency control mechanisms. It also uses the client as the proposer in Paxos. It layers transaction processing over the inconsistent replication of highly-available datastores, and exploits the high availability of data for participant replica recovery. TAPIR also returns the transaction outcome to the client within one processing phase of commit, but the transaction outcome is only visible to other transactions after two phases. Assuming inconsistent replication, TAPIR places strong requirements on application usage

patterns, e.g., pairwise invariant checks and consensus operation result reverse. Replicated commit [35] layers Paxos over 2PC. In essence, it replicates two-phase commit operations among datacenters and uses Paxos to reach consensus on the commit decision. It requires the full replica in each data center, which processes transactions independently and in a blocking manner.

All the above ACPs for highly-available datastores take the vote-after-decide approach. In comparison, HACommit exploits the vote-before-decide approach to enable the removal of one processing phase and the removal of logging in commit. HACommit overlaps the participant voting with the processing of the last operation. Using the unique client as the transaction coordinator and the initial Paxos proposer, HACommit commits the transaction in one phase, at the end of which the transaction data is made visible to other transactions. HACommit exploits the high availability of data for failure recovery, instead of using the classic approach of logging. HACommit is non-blocking under client failures and when fewer than a quorum of participants fail for a transaction.

8 Conclusion

We have proposed HACommit, a logless one-phase commit protocol for highly-available datastores. In contrast to the classic vote-after-decide approach to distributed commit, HACommit adopts the vote-before-decide approach. This approach turns the atomic commit problem into a consensus problem.

In HACommit, the procedure for processing the last transaction operation is redesigned to support the vote-before-decide approach. The last operation processing is then overlapped with the voting process. To commit a transaction in one phase, HACommit exploits Paxos and uses the unique client as the initial proposer. To exploit Paxos, HACommit designs a transaction context structure to keep Paxos configuration information. To ensure that transactions can end with its data visible to other transactions, HACommit has a recovery process for client failures. For participant replica failures, HACommit has participants replicate their votes and the transaction metadata to their replicas; and, a failure recovery process is proposed to exploit the replicated votes and metadata.

Our evaluation demonstrates that HACommit outperforms recent atomic commit solutions for highly-available datastores. In the best case, HACommit can commit in one fifth of the time that the widely used 2PC does.

Acknowledgment

This work is supported in part by the State Key Development Program for Basic Research of China (Grant No. 2014CB340402), the National Key R&D Program of China (No. 2016YFB1000201), the National Natural Science Foundation of China (Grant No. 61303054 and 61420106013), and Youth Innovation Promotion Association of Chinese Academy of Sciences.

References

1. Alipay. Https://www.alipay.com/
2. Baidu wallet. Https://www.baifubao.com/
3. An implementation of the mdcc protocol. Https://github.com/hiranya911/mdcc
4. Paypal. Https://www.paypal.com/
5. Amazon cloud goes down friday night, taking netflix, instagram and pinterest with it (2012). Http://www.forbes.com/sites/anthonykosner/2012/06/30/ amazon-cloud-goes-down-friday-night-taking-netflix-instagram-and-pinterest- with-it/
6. Abdallah, M., Guerraoui, R., Pucheral, P.: One-phase commit: does it make sense? In: Proc. of International Conference on Parallel and Distributed Systems, pp. 182–192. IEEE (1998)
7. Aguilera, M.K.: Stumbling over consensus research: Misunderstandings and issues. In: Replication, pp. 59–72. Springer (2010)
8. Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., Léon, J.M., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing scalable, highly available storage for interactive services. In: Proc. of CIDR, pp. 223–234 (2011)
9. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ansi sql isolation levels. ACM SIGMOD Record **24**(2), 1–10 (1995)
10. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency control and recovery in database systems, vol. 370. Addison-wesley New York (1987)
11. Bodik, P., Fox, A., Franklin, M.J., Jordan, M.I., Patterson, D.A.: Characterizing, modeling, and generating workload spikes for stateful services. In: Proceedings of the 1st ACM symposium on Cloud computing, pp. 241–252. ACM (2010)
12. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: Proc. of OSDI, pp. 335–350. USENIX Association (2006)
13. Chandra, T., Griesemer, R., Redstone, J.: Paxos made live-an engineering perspective (2006 invited talk). In: Proceedings of PODC'07, vol. 7 (2007)
14. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st SoCC. ACM (2010)
15. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Google's globally-distributed database. Proceedings of OSDI p. 1 (2012)
16. Cristian, F.: Synchronous and asynchronous. Commun. ACM **39**(4), 88–97 (1996)
17. Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A.: Remus: High availability via asynchronous virtual machine replication. In: Proc. of NSDI'08, pp. 161–174. San Francisco (2008)
18. Dean, J., Barroso, L.A.: The tail at scale. Commun. ACM **56**(2), 74–80 (2013)
19. Diaconu, C., Freedman, C., Ismert, E., Larson, P.A., Mittal, P., Stonecipher, R., Verma, N., Zwilling, M.: Hekaton: Sql server's memory-optimized oltp engine. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 1243–1254. ACM (2013)
20. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2), 374–382 (1985)
21. Glendenning, L., Beschastnikh, I., Krishnamurthy, A., Anderson, T.: Scalable consistency in scatter. In: Proceedings of SOSP, pp. 15–28. ACM (2011)
22. Goldstein, J., Larson, P.Å.: Optimizing queries using materialized views: a practical, scalable solution. In: ACM SIGMOD Record, vol. 30, pp. 331–342. ACM (2001)
23. Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (2006)
24. Gray, J., Reuter, A.: Transaction processing. Morgan Kaufíann Publishers (1993)
25. Guerraoui, R.: Revisiting the relationship between non-blocking atomic commitment and consensus. In: Distributed Algorithms, pp. 87–100. Springer (1995)
26. Guerraoui, R., Larrea, M., Schiper, A.: Reducing the cost for non-blocking in atomic commitment. In: Proc. of ICDCS, pp. 692–697. IEEE (1996)
27. Guerraoui, R., Schiper, A.: The decentralized non-blocking atomic commitment protocol. In: Proc. of IEEE Symposium on Parallel and Distributed Processing, pp. 2–9. IEEE (1995)
28. Harizopoulos, S., Abadi, D.J., Madden, S., Stonebraker, M.: Oltp through the looking glass, and what we found there. In: Proc. of SIGMOD, pp. 981–992. ACM (2008)

29. Jones, E.P., Abadi, D.J., Madden, S.: Low overhead concurrency control for partitioned main memory databases. In: Proc. of SIGMOD, pp. 603–614. ACM (2010)
30. Kraska, T., Pang, G., Franklin, M.J., Madden, S.: Mdcc: Multi-data center consistency. In: Eurosys (2013)
31. Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16**(2), 133–169 (1998)
32. Lamport, L.: Paxos made simple. ACM Sigact News **32**(4), 18–25 (2001)
33. Leach, P., Mealling, M., Salz, R.: Rfc 4122 - a universally unique identifier (uuid) urn namespace (2005). Internet Engineering Task Force
34. Lee, J., Muehle, M., May, N., Faerber, F., Sikka, V., Plattner, H., Krueger, J., Grund, M.: High-performance transaction processing in sap hana. IEEE Data Eng. Bull. **36**(2), 28–33 (2013)
35. Mahmoud, H.A., Pucher, A., Nawab, F., Agrawal, D., Abbadi, A.E.: Low latency multi-datacenter databases using replicated commits. In: Proc. of the VLDB Endowment (2013)
36. Malviya, N., Weisberg, A., Madden, S., Stonebraker, M.: Rethinking main memory oltp recovery. In: Proc. of ICDE, pp. 604–615 (2014)
37. Mohan, C., Lindsay, B., Obermarck, R.: Transaction management in the r* distributed database management system. ACM Trans. Database Syst. **11**(4), 378–396 (1986)
38. Mu, S., Cui, Y., Zhang, Y., Lloyd, W., Li, J.: Extracting more concurrency from distributed transactions. In: Proc. of OSDI (2014)
39. Mu, S., Nelson, L., Lloyd, W., Li, J.: Consolidating concurrency control and consensus for commits under conflicts. Proc. OSDI (Nov 2016) (2016)
40. Nawab, F., Agrawal, D., Abbadi, A.E.: Message futures: Fast commitment of transactions in multi-datacenter environments. In: Proc. of CIDR (2013)
41. Nawab, F., Arora, V., Agrawal, D., El Abbadi, A.: Minimizing commit latency of transactions in geo-replicated data stores. In: Proceedings of SIGMOD'15, pp. 1279–1294. ACM (2015)
42. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al.: Scaling memcache at facebook. In: nsdi, vol. 13, pp. 385–398 (2013)
43. Pang, G., Kraska, T., Franklin, M.J., Fekete, A.: Planet: making progress with commit processing in unpredictable environments. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 3–14. ACM (2014)
44. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: OSDI, vol. 10, pp. 1–15 (2010)
45. Reynal, M.: A short introduction to failure detectors for asynchronous distributed systems. ACM SIGACT News **36**(1), 53–70 (2005)
46. Schad, J., Dittrich, J., Quiané-Ruiz, J.A.: Runtime measurements in the cloud: observing, analyzing, and reducing variance. Proceedings of the VLDB Endowment **3**(1-2), 460–471 (2010)
47. Shute, J., Vingralek, R., Samwel, B., Handy, B., Whipkey, C., Rollins, E., Oancea, M., Littlefield, K., Menestrina, D., Ellner, S., Cieslewicz, J., Rae, I., Stancescu, T., Apte, H.: F1: A distributed sql database that scales. Proc. VLDB Endow. **6**(11), 1068–1079 (2013)
48. Skeen, D.: Nonblocking commit protocols. In: Proc. of SIGMOD, pp. 133–142. ACM (1981)
49. Sovran, Y., Power, R., Aguilera, M.K., Li, J.: Transactional storage for geo-replicated systems. In: Proc. of SOSP'11, pp. 385–400
50. Stamos, J.W., Cristian, F.: Coordinator log transaction execution protocol. Distributed and Parallel Databases **1**(4), 383–408 (1993)
51. Zhang, I., Sharma, N.K., Szekeres, A., Krishnamurthy, A., Ports, D.R.: Building consistent transactions with inconsistent replication. In: Proceedings of SOSP '15. ACM, New York, NY, USA (2015)