2016 IEEE 22nd International Conference on Parallel and Distributed Systems

Catenae: Low Latency Transactions across Multiple Data Centers

Ying Liu, Qinjin Wang and Vladimir Vlassov
Department of Software and Computer Systems
KTH Royal Institute of Technology, Sweden
Email: yinliu@kth.se, wangqinjin@gmail.com, vladv@kth.se

***Abstract*—Serving requests with low latency while data are replicated and maintained consistently across large geographical areas, e.g. in multiple data centers (DCs), is challenging. We propose Catenae, a transaction framework that provides serializable transaction support for data replicated in multiple DCs. Catenae leverages periodic replicated epoch messages to reduce synchronization delay among DCs, which results in the reduction of commit latency of transactions. It employs and extends a transaction chain concurrency control algorithm to speculatively execute transactions in DCs with maximized execution concurrency and determinism of transaction ordering. As a result, Catenae is able to commit a transaction with half a RTT to a single RTT across DCs in most of the cases. Evaluations with TPC-C benchmark have shown that Catenae significantly outperforms Paxos Commit over 2-Phase Lock and Optimistic Concurrency Control. Catenae doubles the throughput and halves the commit latency comparing to the both approaches.**

Keywords*—*Transaction; Speculative Transaction; TPC-C; 2PL; OCC; Geo-replicated; Geo-distributed; Distributed Storage System

I. Introduction

Nowadays, data tend to be served to clients all over the world. Large scale web applications, such as Facebook and Twitter, tend to host and serve data from multiple DCs. This increases service availability by allowing systems to survive with complete outage of DCs, that can be caused by technical problems [1], unforeseen natural disasters [2], or a sudden surge in traffic [3]. Furthermore, spanning services across multiple DCs improves service latency since clients can access data from a nearby DC. However, it is challenging for a database system to realize these advantages considering the high latency across DCs. Data partitions involved in a transaction could be hosted in different DCs, that incurs high communication overhead to maintain ACID properties using traditional concurrency control algorithms, such as two phase lock (2PL) [4] and optimistic concurrency control (OCC) [5]. Furthermore, high availability and low latency need data replication. Maintaining data consistency among replicas in multiple DCs also involves a large amount of cross DC communications.

In order to address these challenges, we investigate the triggers of cross DC communications. In essence, these communications are mainly used for synchronizations in two scenarios. First, algorithm maintains the total ordering of different transactions with respect to different data partitions. Second, algorithm tackles with the ordering of operations on replicas for maintaining replica consistency.

We propose a framework, Catenae, which provides serializable transactional support for a data store deployed in multiple DCs. It manages the **concurrency among transactions** and maintains **data consistency among replicas**. In order to reduce cross DC synchronizations, Catenae leverages the insight of using speculative executions of transactions in each DC, which expects a coherent total ordering of transactions in all DCs and eliminates the need for synchronizing replicas.

However, efficiently achieving speculative executions of transactions with a deterministic order on a global scale is not trivial. Static analysis of transactions before execution is able to produce a deterministic ordering among transactions. Nevertheless, this approach has the disadvantage of high static analysis overhead and potentially inefficient scheduling among transactions. To be specific, a complete set of transactions needs to be analyzed and ordered at a single site in the system, which is a scalability bottleneck and a single point of failure. Moreover, static ordering of transactions cannot guarantee efficient executions in terms of concurrency. Because it is impossible to efficiently order conflicting transactions when their access time on each data partition is unknown before execution. Approaches, such as ordering transactions by comparing the receiving timestamps [6], lead to inefficient execution of transactions for the same reason.

Catenae implements and extends a Transaction Chain (TC) concurrency control algorithm [7] to maximize the determinism of transaction execution orders among multiple DCs. TC orders transactions based on their access time on each data partition to maximize concurrency. To give an overview, Catenae schedules the same set of transactions to be executed in each DC in fine-grained periods of time using a novel epoch boundary protocol (Section V). Then, batches of the same transactions are executed using TC in each DC without cross DC synchronization. Catenae speculates the same execution order of transactions under TC concurrency control in each DC. Having a homogeneous setup in each DC can maximize the determinism. The speculative executions are then validated by a second phase Paxos [8].

We compare the performance of Catenae with Paxos Commit (PC) [9] over Two-Phase Lock (2PL) [4] and PC over Optimistic Concurrency Control (OCC) [5] under TPC-C. Catenae achieves more than twice of the throughput than 2PL and OCC with 50% less commit latency. Both 2PL and OCC observe significant abort rate under saturating read-write transactional workload while it does not happen in Catenae.

In summary, our contributions are the following:

- We design and implement an epoch boundary protocol that is used to synchronize information among DCs.
- We provide distributed protocols to efficiently coordinate, execute and commit transactions.The average execution time for a read-write transaction is slightly larger than a round trip time (RTT) among DCs.

- We implement Catenae, a low latency transaction framework for geo-replicated data stores.
- We evaluate Catenae against PC over 2PL and PC over OCC under TPC-C benchmark. The results indicate that Catenae significantly outperforms both approaches in terms of transaction throughput, latency and commit rate.

II. Related Work

Data Replication. Initially, data is stored in a replicated fashion to improve its availability [10], [11]. Then, the technique of data replication is also used to improve service latency [11], [12], especially tail latency [13], since the fastest response from any replica can be returned to clients.

Geo-replication. Nowadays, many services are serving clients all over the world. Systems tend to replicate data in different DCs to have a wide geographical coverage in order to service data close to clients and provide better service availability. Google Spanner [14] is one of the representative system designed for serve data geographically. It benefits from low read request latency since data are served locally. However, maintain data consistency across a large wide-area involves significant communication overhead, since across DC links are with large transmission delays. Recent works [15], [16], [14], [17], [6] optimizes the frequency of using cross DC communications to keep data consistent.

Global Time. Having a global knowledge of time helps to reduce the synchronization among replicas since operations can be naturally ordered based on global timestamps. However, synchronizing time in distributed systems is extremely challenging [18], which leads us to the application of loosely synchronized clock. It is adapted in many recent works to achieved different consistency models from casual consistency [19], [20], [21] to linearizability [14].

Transaction Support. Previously, transactions are supported by traditional database systems where data are not replicated. To support transactions in a large scale on top of a storage system where data are widely replicated is challenging. There are geo-distributed transaction frameworks that are built on replicated commit [15], [17], paxos commit [16], [14], and deterministic total ordering based on prior analysis of transactions [22], [23].

Catenae is a transaction framework for geo-distributed data stores. It differs from the existing approaches in two ways. First, it extends transaction chains [7] to achieve deterministic execution of transactions without prior analysis of transactions. This improves transaction execution concurrency, removes bottleneck and single point of failure. Second, a novel epoch boundary protocol is designed and implemented to coordinate transaction executions in multiple DCs with reduced RTT rounds. Updates are not requested (requires a RTT) but rather actively propagated (requires half a RTT) under epoch boundary protocol among DCs.

Comparing to the original transaction chain algorithm proposed in [7], Catenae extends the algorithm for replicated data stores. Specifically, Catenae allows multiple versions of a record in chain servers to enable read-only transactions and support transaction catch ups in case of replica divergence. The extended transaction chain algorithm manages the concurrency among transactions in the same DC while the epoch boundary protocol controls the execution of transactions among DCs.

III. Challenges

Achieving low latency transactions while maintaining serializability in a geo-distributed environment is not trivial. The time spent from receiving a transaction until it is committed and returned to the client defines the latency of a transaction. A transaction may need to obtain consensus from data replicas stored in other DCs before committing. This process involves message **transmission delays** among DCs and **concurrency delays** to reach a consensus on a serializable execution order of conflicting transactions in all DCs. When consensus is reached, a transaction is executed with an **execution delay**.

The transmission delay depends on the locations and connectivity of DCs and usually cannot be optimized. Reducing the message exchanging rounds among DCs to reach a consensus on the execution of transactions are studied in recent works [15], [16]. It is theoretically proved that the lower bound for two conflicting transactions to commit and maintain serializability in two DCs is the RTT between them [6].

The concurrency delay is the time spent for a transaction to be allowed to commit in all DCs. The concurrency delay is caused by conflicting transactions. Examples of the concurrency delay can be the waiting time for locks in 2PL or the time spent to abort and retry in OCC.

The execution delay is subjective to the competence of the hosting platform and the efficiency of the underlying storage system while performing read and write requests.

A. Proposal

Catenae executes transactions with low latency by improving the transmission and concurrency delays. In order to reduce message synchronizations among DCs, it first speculatively executes transactions in each DC and validates later to commit transactions executed with the same dependency in all DCs.

Many previous works [23], [22] achieve this by analyzing transactions before execution and giving priority to some transactions while aborting or suspending conflicting transactions, in order to have only non-conflicting transactions to be executed in parallel on data replicas. In essence, the concurrency control in those approaches is similar to two phase locking (2PL), which increases the transaction execution time and limits the throughput. For example, a transaction T_1 arrives at t_1 and writes on data partition a and b while another transaction T_2 arrives later at $t_2(t_2 > t_1)$ and writes on data partition b. T_1 and T_2 are conflicting with each other and a total order needs to be preserve on all replicas of data partition a and b in order to maintain serializability. Usually, a static analysis before the execution is hard to know which transaction should have the priority to be executed first. Typically, such priority is given based on the arrival time of transactions. Thus, T_1 is ordered before T_2. Assuming the time spent on writing each data partition is constant Δt, then, the execution time of T_1 and T_2 is $2 * \Delta t + \Delta t = 3 * \Delta t$. Obviously, this type of concurrency controls can potentially block the concurrency of transaction executions.

Catenae pushes transaction execution concurrency to the limit by delaying the decision on transaction execution orders until they are conflicting a shared data partition. This allows transactions to be ordered naturally by their execution speed rather than their arrival time. Back to the example, assuming T_2 arrives slightly behind T_1, which gives $t_2 - t_1 < \Delta t$, T_2 is able to access data partition b before T_1 since T_1 has not finished writing on data partition a. When T_1 has finished writing on

a and continues to write on b at time $t_1 + \Delta t$, it observes that T_2 is in the middle of writing on b. Then, T_1 is naturally ordered behind T_2 and will write on b until T_2 finishes. The total execution time of T_1 and T_2 is $\Delta t + t_2 - t_1 + \Delta t = 2 * \Delta t + t_2 - t_1 < 3 * \Delta t$. A formalization of this concurrency control is a **transaction chain concurrency control** algorithm, which will be explained in details in Section VI.

The insight in Catenae is that the execution speed of transactions on each record is unique and deterministic. Ideally, Catenae believes that given the same set of transactions to multiple **fully replicated DCs**, the execution order of the conflicting transactions in these DCs are likely to be the same using the transaction chain concurrency control. Experimental validations (Section III-B) and evaluations (Section VII) of Catenae under a symmetric cluster setup, i.e. the same VM instance type in multiple DCs on top of Google Cloud Platform, shows that most of the conflicting transactions are ordered identically in all DCs. Thus, Catenae first speculatively executes the same set of transactions in each DC. Then, inconsistent executions will be corrected by a validation phase.

B. *Validations of the insight*

We validate the success rate of speculative executions of Catenae in three DCs of Google Cloud Platform. Specifically, we have randomly generated 10000 records and replicated them on 4 storage servers in each DC. These random records have different data sizes, which leads to different access times when reading or writing on the records. Then, we have a coordinator in each DC that generates transactions with specific throughput to the 4 storage servers in the same DC. We guarantee that coordinators generate the same transaction sequence with Poisson arrivals. Each transaction will read/write 1 to 4 data records out of 10000 records. The distribution of the record accessed is configured to be uniform random, zipfian with exponent 1 or zipfian with exponent 2. Figure 1 presents the evaluation of running 100000 transactions in each DC. Those transactions are generated to the storage servers with different rates, which are from 1000 to 11000 request per second as shown on the x axis. Storage servers execute transactions using transaction chains concurrency control algorithm. In short, the algorithm orders transactions based on the access order on the first shared data record. A simple example of the algorithm is presented in the previous section and the detailed explanation of the algorithm will be discussed in Section VI The execution dependencies of each transaction in each DC are compared. If the execution dependency of a transaction is the same in all three DCs, it means a success in speculative execution. Otherwise, the speculative execution is invalid. The y-axis in Figure 1 illustrates the success rate of speculative execution under 3 workload access patterns, i.e., uniform random, zipfian with exponent 1 and zipfian with exponent 2. The results indicate that the transaction chain algorithm is able to allow transactions to be executed on record replicas without coordination but still yields a very high (above 80%) result consistency rate (success rate of speculative execution) when the access pattern of the workload is uniform. Even when the access pattern is zipfian with exponent 1, Catenae is able to obtain a reasonable success rate (above 60%) on speculative execution using transaction chains. However, the evaluation results also show that the speculative execution will fail with extremely contended access pattern (zipfian with exponent 2).

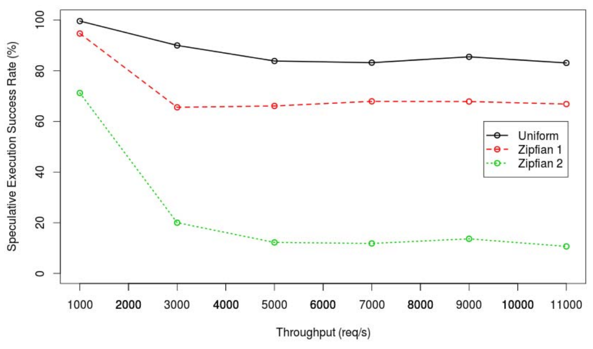

Fig. 1. Success rate of speculative execution using transaction chain

IV. The Catenae Framework

Transaction Client. Catenae has a transaction client library for receiving and pre-processing transactions. Transaction clients are the entries of Catenae in each DC. They pre-process transactions from standard query languages, such as SQL, and chop them to sequences of key-value read/write operations.Then, pre-processed transactions are forwarded to coordinators for scheduled execution among DCs.

Coordinator. There is one coordinator in each DC, which is responsible for the speculative executions and validations of transactions among DCs. It is achieved through the exchange of epoch messages among coordinators in different DCs in a fixed time interval. We defer the explanation of epoch messages in Section V Coordinators are designed to be stateless in each DC, thus Catenae can have multiple coordinators in one DC by partitioning the responsible namespace range.

Secondary Coordinator. There is an optional secondary coordinator for each DC that stands by the coordinator in that DC. Secondary coordinator receives duplicated epoch messages from other coordinators. It becomes primary coordinator when the coordinator fails.

Chain Servers. Transaction chain servers are hosted together with storage nodes of a NOSQL data store. Transaction chain servers are responsible for traversing of a transaction chain by passing through its forward and backward pass phase, during which, it maintains and resolves transaction dependencies and conflicts, record temporary copies of transaction execution results, and issues corresponding NOSQL operations to the underlying storage servers when the transaction commits. The transaction chain algorithm is explained in Section VI

Transaction Resolver. There is a transaction resolver in each DC. It maintains implicit dependencies to avoid cyclic dependency among transactions. It is queried by transaction chain servers when they suspect a formation of a circle during transaction execution. Transaction resolvers perform a topology sorting among transactions with respect to the existing explicit dependencies. Then, a circle-free implicit dependency is returned to the chain server and stored in the transaction resolver for further queries until the involved transactions have committed or aborted.

V. Epoch Boundary

Epoch boundary is a concept similar to logical clock proposed by Lamport, but using the real time from the system. It separates continuous time into discrete time slices. The start or end of a discrete time slice is a boundary. Time boundaries are used as synchronization barriers among replicated servers deployed in different DCs. In Catenae, synchronizations of the status of replicated servers are not triggered by events, such as a transaction is received by one server or a consensus is needed to validate an execution result, but rather is conducted periodically at each boundary. The advantage of actively synchronizing server states among DCs is that it reduces the delay for a DC to realize the updates from other DCs. Specifically, when a DC needs additional information to proceed an operation, for example, to validate whether it holds the most recent data copy, it does not need to send a request and wait for a response to/from another DC, but rather wait for the next epoch boundary. It optimizes the communication latency among DCs from a RTT to a single trip plus the delay of an epoch. Epoch boundaries are not suitable to be implemented in low latency networks, such as intra DC networks, when inter-server latency is low. In this case, a RTT is rather short comparing to an epoch. Furthermore, periodically sending and receiving epoch messages also involves non-trivial overhead. However, this approach prevails when servers need to communicate through high latency links, such as inter DC links, when an epoch delay is negligible comparing to a single message trip. Specifically, the typical RTTs among DCs are from 50ms to 400ms, which can be easily measured through [24]. In contrast, the typical setup of the epoch interval is from 5ms to 30ms.

As shown in Figure 2, $DC2$ is able to aware an event happened at t in $DC1$ with delay less than $C + E$. However, with active queries, $DC2$ will know the status of $DC1$ after a delay of $2 * C$, which is significantly larger than $C + E$.

In order to ease the maintenance of server membership and reduce the overhead of sending epoch messages, there is one coordinator server in each DC to maintain epoch boundaries. Time in each coordinator server is synchronized using NTP to minimize the time drifts. The length of the epochs is a globally configurable parameter. Epochs are associated with monotonically increasing epoch IDs that is coherent on each coordinator. At the end of each epoch, a synchronization boundary is placed with the dispatching of status updates (payload) from/to all coordinators using epoch messages.

A. *Transaction Distribution Payload*

The first part of epoch payload relates to transaction distribution. Ideally, with epoch boundaries, each DC is able to acquire the transactions received from other DCs with a single cross DC message delay plus an epoch. By knowing the complete set of input transactions in an epoch, Catenae can speculatively execute transactions using the TC algorithm (Section VI-C) and maximize the possibility to obtain a coherent execution order of conflicting transactions in all DCs.

B. *Transaction Validation Payload*

The second part of epoch payload concerns about transaction validation. The speculative executions need to be validated on the execution order of conflicting transactions in all DCs since they can be executed in different orders. Catenae leverages a light-weight static analysis of input transactions to create different transactions sets. The transaction set with conflicting transactions needs a validation phase to confirm their execution results. We defer the explanation of the multiple DC transaction chain algorithm in Section VI.

C. *Batching and Dispatching of Payloads*

For transaction distribution payload, all coordinators batch transactions received in each epoch. These transactions are associated with a local physical timestamp when it is received by Catenae. For transaction validation payload, coordinators batch conflicting transactions that have finished in each epoch along with their execution dependencies. The batched payloads are sent among coordinators at the end of each epoch. Instead of simply exchanging the payload of the current epoch, coordinators also attach the payload from the previous two epochs. According to our experiments, the redundancy in epoch payloads effectively handles message losses and delays during transmission among coordinators.

VI. Multi-DC Transaction Chain

The life cycle of an transaction in Catenae includes received, scheduled, executed, finished, and committed (returned). We present the multi-DC transaction chain algorithm with the explanation of the life cycle of a transaction.

A. *Receive Transactions*

Coordinators receive transactions from other coordinators in epochs. Due to the transmission delays, transactions received in the current epoch are transactions sent by other coordinators in a past epoch. For example, in Figure 2, transactions received by $DC2$ at e_y are transactions sent from $DC1$ at e_x. The epoch ID (EID) is used to identify an epoch message. Coordinators continuously receive and keep track of epoch messages from other DCs and aggregate them by EIDs. With the complete receipt of epoch messages from all the coordinators concerning the same EID, the transactions in the epoch messages are grouped together and moved to transaction schedule phase. Transactions in lower EIDs are scheduled before transactions in higher EIDs. This allows Catenae to have a more consistent execution of transactions in each DC. However, it also puts limitations on Catenae when there are failures, which is discussed in Section VIII.

B. *Schedule Transactions*

Transactions are chopped into a set of read and write operations by Catenae client library. Read and write operations of a transaction are ordered deterministically based on the accessed data partitions. The data partitions are traversed in the deterministic order monotonically by the transaction chain algorithm. Specifically, operations are mapped to Catenae chain servers that store the corresponding data partitions. Then, the transaction is sent to the chain server that hosts the first accessed data partition to start traversing. Thus, Catenae does not support transactions that have cyclic or conditional accesses on any data partitions.

C. *Execute Transactions*

Transaction execution in each DC is handled by a transaction chain (TC) concurrency control protocol. It allows concurrent transactions to commit freely in the natural arrive order on the storage servers unless doing so will violate serializability. This property maximizes the transaction execution concurrency by allowing transactions to execute based on their execution speed and wait only if a faster transaction already occupied the resources on a per-key granularity. This means that transaction execution is not based on a predefined order given by the prior static analysis [23], [22] or the arrival order, but the access

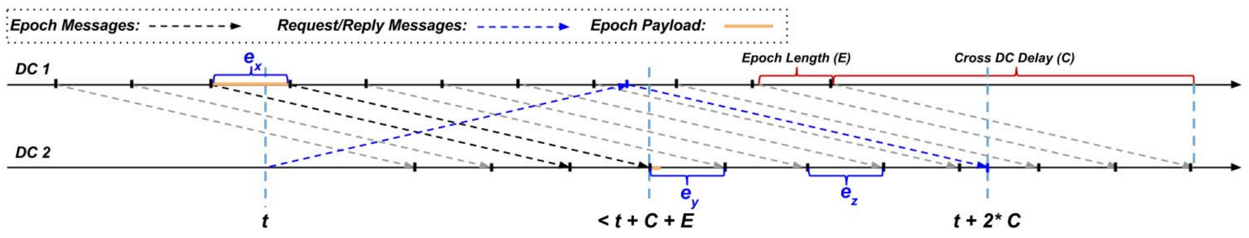

Fig. 2. Epoch Messages among Data Centers

order at a shared data partition where two transactions issue conflict operations. Since transactions are executed in DCs individually, we explain the TC algorithm from the perspective inside one DC. The algorithm needs to pass through two phases, i.e., forward pass and backward pass.

1) Forward pass: The forward pass does not conduct any read/write operations, but rather leaves footprints of a transaction on accessed data partitions. These footprints are used to identify conflicting transactions. It starts with the coordinator sending a transaction to the first accessed chain server as specified in the chain. The chain server records the data partitions that the transaction reads or writes, then the transaction is forwarded to the next chain server specified in the chain until reaching the end of the transaction chain.

2) Backward pass: When a transaction is on the last server of its transaction chain during the forward pass, it starts the backward pass phase. The backward pass examines whether other transactions that have left footprints and have pre-committed values on the accessed data partition. If not, the transaction may read or pre-commit on the data partition. Otherwise, the pre-committed transactions are added as dependent transactions of the current transaction. The following backward pass of the transaction needs to strictly obey the dependency, i.e. ordering behind the dependent transactions. It summarizes as the first execution rule. Specifically, for example, in Figure 3, T_1 has conducted backward pass and pre-committed a write on partition $S_5 : k_5$ before T_5. So, T_5 adds T_1 as its dependent transaction. Then, T_5 backward passed to $S_1 : k_1$ before T_1. T_5 knows T_1 will access $S_1 : k_1$ because it has left a footprint on $S_1 : k_1$ during its forward pass. Thus, T_5 needs to wait for T_1 on $S_1 : k_1$ even it arrives earlier in order to maintain serializability on $S_1 : k_1$ and $S_5 : k_5$.

Rule 1: A transaction depends on another transaction if it comes later to the first shared partition in its backward pass. And the transaction is consistently ordered after transactions that it depends on regarding all the shared partitions afterwards.

In addition to explicit dependencies added by Rule 1, a transaction also has to satisfy a set of implicit dependencies. Implicit dependencies are added to a transaction to prevent cyclic dependencies. For example, in Figure 3, according to Rule 1, T_3 is ordered after T_4 when accessing $S_3 : k_3$. And T_4 is ordered after T_1 when accessing $S_2 : k_2$. Transitive relation gives T_3 should be ordered after T_1, otherwise a cyclic dependency will form. However, without any hints, T_3 could be ordered before T_1 when it arrives faster on $S_1 : k_1$.

Implicit dependencies are added by a transaction resolver, which has a global view of potentially conflicting transactions in all chain servers. Detecting complete cyclic behaviors could be a NP-hard problem. Our resolver uses the pattern shown in Figure 4 to detect potential cyclic behaviors, which is proved to be effective and efficient in detecting cyclic dependency in transactions [25]. A topology sorting request is sent from a chain server to the resolver, when the above pattern is captured. The resolver provides a serializable sorting of the transactions that does not violate the observed constraints recorded in its transaction dependency repository. The repository stores the dependencies that have been observed by the resolver from requests sent by other chain servers. Then, the sorting result is returned to the chain server and recorded in the transaction dependency repository for future queries.

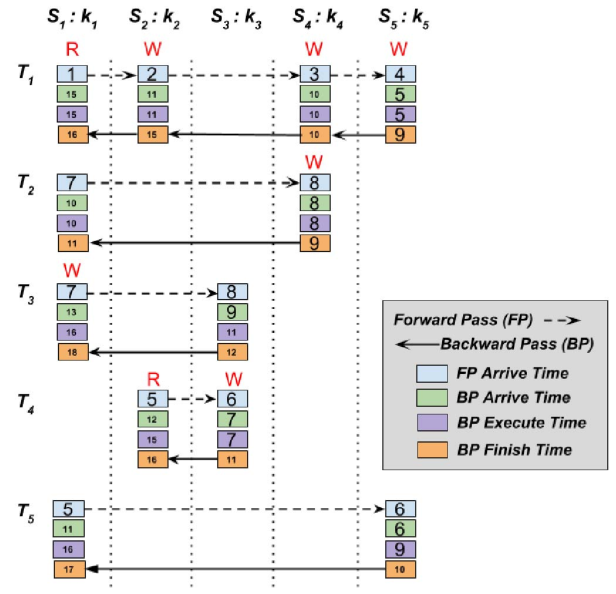

Fig. 3. An example execution of transactions under transaction chain concurrency control

Continuing the above example in Figure 3, T_4 knows that it is ordered before T_3 on $S_3 : k_3$. and when it knows that it is ordered after T_1 on $S_2 : k_2$, the pattern in Figure 4 forms. So, S_2 requests a topology sorting to the resolver. The resolver returns the only serializable topology sorting T_3 ordered after T_1. When T_3 passes to $S_1 : k_1$, the pattern also forms because it has dependency with T_4 and about to have dependency with T_1. So, $S_1 : k_1$ queries the resolver, which will return the already calculated constraint in its repository, which is $T_3 \Rightarrow T_1$. So, T_3 waits for T_1 on $S_1 : k_1$.

When a transaction has acquired both the explicit and implicit dependencies, it attempts to read/write temporary values on a chain server, which is the second execution rule.

Rule 2: If all dependent transactions have already pre-committed or aborted on the particular chain server, the current transaction is able to pre-commit. Otherwise, the transaction needs to wait until the condition is satisfied.

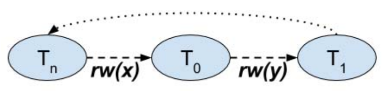

Fig. 4. Potential Cyclic Structure

D. Validate Transactions

Since Catenae speculatively executes transactions in all DCs using TC without synchronization, conflicting transactions can be executed with different dependencies. Speculative executions with validations tradeoff Catenae's percentile performance for its average performance.

1) Non-conflicting Transactions: If transactions access data partitions that are solely accessed by themselves, there is no need for transaction validations since the commit orders among these transactions can be different in different DCs while serializability is still preserved. These transactions are non-conflicting. A transaction that is not conflicting with the others when the accessed data partitions are not accessed by other transactions until the end of this transaction's backward pass. A transaction can finish its backward pass at different times in different DCs and this may cause inconsistent judgement on whether the transaction is non-conflicting or conflicting. To solve this issue, a priority is given to the DC where the transaction is initiated since it will be the DC that returns the execution result to the client. If this DC decides a transaction to be non-conflicting, then it will skip the validation phase and return the results to the client directly. In this case, a dominant result will be propagated to other DCs to commit the transaction.

2) Conflicting Transactions: Conflicting transactions need to reach a consensus among DCs on their execution dependencies. The execution results and the execution dependencies are part of the payload in the epoch messages as described in Section V. Upon receiving the majority execution results of a transaction from other coordinators, a second phase of the Paxos algorithm [8] is executed independently on all coordinators. Specifically, when there are a majority of DCs that have executed the transaction with the same dependency, then the transaction will be committed with this dependency. DCs that have executed the transaction with this dependency prepare to commit the temporary read/write operations from chain servers to their underlying storage servers. DCs that have performed the transaction with other dependencies will need to perform a catch up procedure explained below.

E. Commit Conflicting Transactions

When a transaction is allowed to commit in a DC, it checks whether its dependent transactions are committed or aborted. If all its dependent transactions are committed, the transaction is able to commit by choosing a commit timestamp from the intersect of decision periods from all DCs. The decision period is a period of epochs when all DCs are expected to received the execution result of a transaction. The lower bound of a decision period is calculated using the current epoch plus an estimated message delay. The upper bound of the decision period is the lower bound plus an offset, which denotes the maximum delay that can be tolerated during message transmission among DCs. The decision period of a transaction from different DCs might be slightly different because of the difference of the execution environment. We deterministically choose the maximum epoch of the intersect of the decision periods from all DCs to tolerate possible delays on the arrivals of the execution results from other DCs. Then, the transaction commit message is sent to all the involved chain servers and, in the meantime, the transaction is returned to the client. Chain servers that have received commit messages from the coordinator commit the operations from the transaction to the underlying storage servers and remove the corresponding dependencies.

If there are uncommitted dependent transactions the pre-committed transaction has to wait until the dependent transactions are committed, caught up or aborted. In this case, the commit epoch may increase beyond the decision period and is deterministically chosen to be the next epoch of the last committed dependent transaction. If the dependent transactions need to catch up, the transaction will need to catch up as well, since it is executed with a super-set dependency. If the dependent transactions are aborted, then the transaction is able to commit if the transaction only write-dependent on the shared key, otherwise, the transaction is aborted as well.

Transaction Catch Up. DCs that have executed a transaction with a different dependency from the majority dependency need to catch up its execution. The catch up of a transaction is executed when all its dependent transactions are committed, aborted or caught up. The catch up procedure applies update operations in a transaction with the majority voted timestamps to the underlying storage system.

Transaction Abort. Transactions can be aborted for various reasons. For example, aborts are issued by Catenae when no majority can be reached on the execution results from all DCs. Aborting a transaction removes its dependency and temporary updates on the chain servers and the resolver.

F. Read Only Transactions

The advantage of a geographically distributed transactional storage system is its ability to serve data close to its clients, which achieves low service latency. In order to achieve that, it is essential to support transactions that can be executed and returned locally. Catenae allows read only transactions to be executed locally while still maintaining ACID property.

Read only transactions are processed by reading values concurrently from the corresponding chain servers. They can be returned when it is not in the decision period of a transaction with uncommitted write on a read data partition. Since all transactions with write operations are committed by choosing the largest possible timestamp of the intersect of decision periods from all DCs, it is safe to read values from the underlying storage servers before the lower bound of a decision period. If the read only transaction has read a data partition during the decision period of an uncommitted transaction that has uncommitted writes, it will retry after a short delay.

VII. Evaluation

The evaluation of Catenae is performed on Google Cloud with three DCs. The performance of Catenae is compared against Paxos commit [9] over Two-Phase Lock (2PL) [4] and Paxos commit over Optimistic Concurrency Control (OCC) [5]. The evaluation of Catenae focuses on performance metrics including transaction commit latency, execution concurrency (throughput) and commit rate. We measure the performance of Catenae under different workload compositions and setups to explore its most suitable usage scenarios using a microbenchmark and standard TPC-C benchmark.

A. Implementation

Catenae is implemented with over 15000 lines of Java code. Chain servers and coordinators are implemented as state machines. They employ JSON to serialize data and Netty sockets to communicate among chain servers and coordinators.

2PL and OCC implementation. Two-phase Lock is implemented by using Paxos commit for managing data replication among DCs and two-phase lock inside DCs to avoid conflicts. There is a coordinator in each DC to manage the lock table and synchronize data replicas when transactions commit. During transaction execution, the coordinator acquires locks and issues temporary writes of the involved data partitions to corresponding data servers (first phase of Paxos commit). The coordinator is able to lock a data partition when the majority of DCs are able to lock the data partition. During transaction commit, the coordinator issues commit messages to other DCs. A transaction is committed when a majority of DC commit (second phase of Paxos commit). Wound-wait mechanism [26] is used to avoid deadlocks.

Optimistic concurrency control also cooperates with Paxos commit to manage data replicas. There is a coordinator in each DC to validate the execution results and synchronize data replicas when transactions commit. Transactions are distributed and executed in all DCs with records on the versions of data partitions that have been read and written. Temporary values are buffered on data servers. Temporary execution results with versions of accessed partitions are voted and validated among coordinators (first phase of Paxos commit). A transaction commits when a majority of DC commit (second phase of Paxos commit). Our OCC implementation allows aborted transactions to retry one time before returning aborts to clients.

B. Cluster Setup

Our evaluations are conducted using Google Cloud Compute Engine. Specifically, Catenae, 2PL and OCC systems are deployed in three DCs, i.e. europe-west1-b, us-central1-a and asia-east1-a. Inside each DC, four Cassandra nodes are used as storage backend running on Google n1-standard-2 instances, which have 2 vCPUs and 7.5 GB memory. Each DC has an isolated Cassandra deployment since data replication is already handled. Catenae chain server, 2PL server daemon and OCC server daemon are deployed on the same servers as Cassandra nodes. Committed writes are propagated to Cassandra using the write-one interface. A Google Cloud n1-standard-8 instance (8 vCPUs and 30 GB memory) is initiated in each DC to serve as a coordinator in all three systems. For Catenae, transaction resolver is configured together with coordinator. A Google Cloud n1-standard-16 instance (16 vCPUs and 60 GB memory) is spawned in each data to run the workload generator. The workload generator propagates workload to services deployed in the same DC. Another n1-standard-16 instance is spawned in each data to serve as frontend client server.

Configuration of Catenae. The epoch length in Catenae is configured as 10 ms, which yields reasonable tradeoff between coordinator utilization and transaction synchronization delays as later shown in Figure 6.

C. Microbenchmark

We implement a workload generator that is able to generate transactions with different number of accessed partitions, different operation types (read/update/insert) and different distribution (Uniform/Zipfian) of accessed partitions. Under different workload compositions, we evaluate the performance of Catenae and the results are compared with 2PL and OCC. Then, an evaluation on the effect of varying epoch length in Catenae is also presented.

Workloads. We evaluate with two types of transactional workloads, i.e. read-only and read-write, with a namespace of 100000 records. Read workload is constructed with transactions that only read on data partitions. Read-write workload is formed with transactions that read, write or update data partitions. An update is translated to a read followed by a write on the same data partition. Each transaction randomly embeds one to five data partitions to be accessed. The access pattern of the involved data partitions can be uniform or zipfian with the exponent equals to one.

1) Results: Figure 5 shows the evaluation results of Catenae, 2PL and OCC under read-only and read-write transactional workloads with uniform and zipfian data access pattern. We use commit latency, throughput and abort rate as performance metrics. The results shown in Figure 5 are the aggregated values from three DCs.

The performance of Catenae, 2PL and OCC is comparable under uniform read-write workload. Under this workload, all three approaches need to synchronize data replicas with remote DCs but transactions are not likely to conflict with each other since the data access pattern is uniform. Catenae outperforms 2PL and OCC because of the application of epoch boundary protocol and the separation of conflicting and non-conflicting transaction sets. They have enabled Catenae to commit non-conflicting read-write transactions with a little more than a half RTT and commit conflicting read-write transactions with slightly more than a single RTT.

The throughput of 2PL and OCC start to struggle and plateau with the increasing number of clients under zipfian read-write workload, where transactions are likely to conflict with each other. As expected, OCC observes significant abort rate under this workload. On the other hand, Catenae scales nearly linearly under both uniform and zipfian workload. This is because of the efficient scheduling of concurrent transactions using the transaction chain concurrency control. Specifically, transactions are not contended until they begin accessing a shared data partition concurrently. Only at this point, the execution dependencies are established. Even so, transactions are allowed to proceed and commit given that the established dependencies are preserved. In sum, the speculative execution using transaction chains achieves very high success rate even under zipfian (with exponent=1) workload as validated in Section III-B (Figure 1).

The performance of 2PL and OCC under read-only workload is similar to their performance under uniform read-write workload since both workloads requires 2PL and OCC to synchronize replicas in remote DCs. The only different is that there is no conflict while executing and committing transactions, which leads to a higher throughput and lower commit latency in both approaches. In contrast, Catenae observes more than three times performance gains in both latency and throughput since read-only transactions can be processed locally in Catenae. It is enabled because the lower-bound of EID that a write-involved transaction is scheduled to be committed is known when it enters the validation phase, which requires a proposal of a decision period (Section VI-E). Thus, it is safe to return a read-only transaction locally when its timestamp is lower than the lower-bound of the decision period

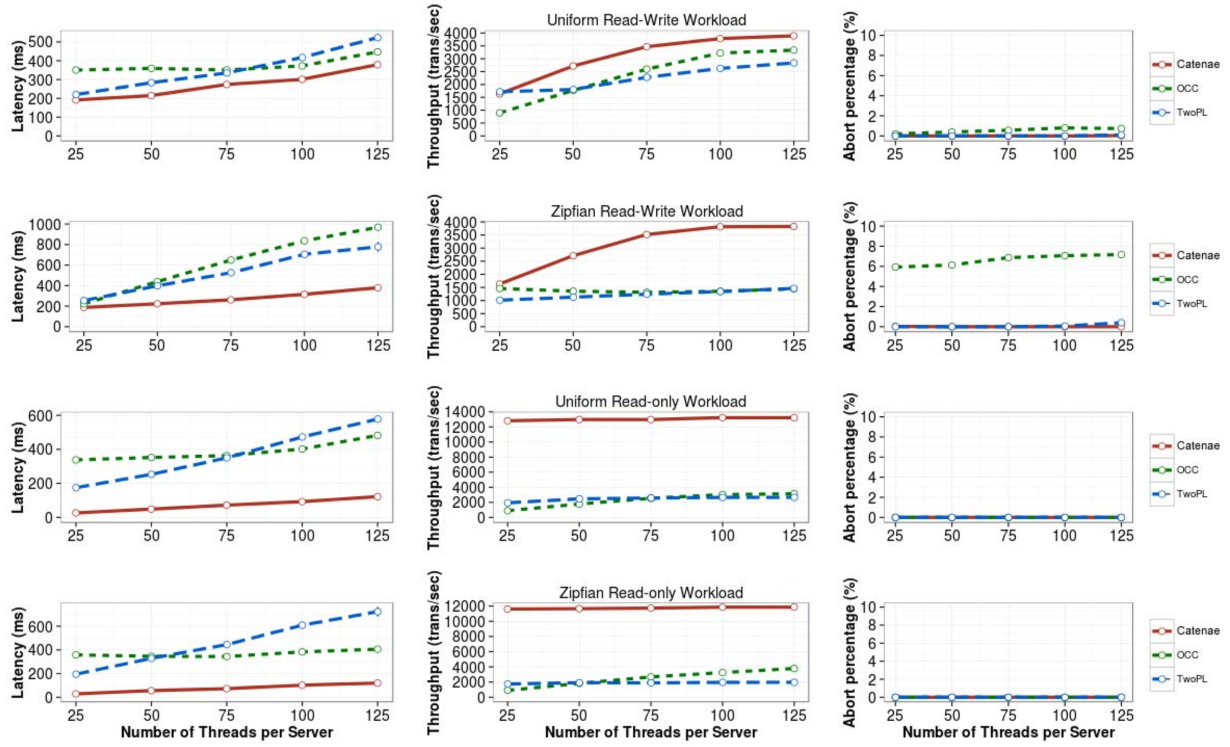

Fig. 5. Performance results of Catenae, 2PL and OCC using microbenchmark

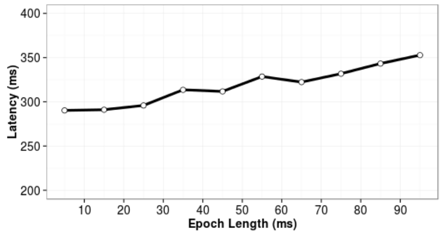

Fig. 6. Commit latency VS. varying epoch lengths using 75 clients/server under uniform read-write workload

of a write-involved conflicting transaction. Otherwise, the read-only transaction is retried after 50 ms.

Varying the Epoch Length. To further study the performance of Catenae, the varying size of epoch length is evaluated. As shown in Figure 6, transaction commit latency starts to increase steadily with epoch length more than 20 ms. This is because that transactions will only be executed after they are propagated to all DCs. The longer the epoch length, the more delay is imposed on transactions. However, too short epoch length leads to frequent exchanging of epoch messages among coordinators, which introduces performance bottleneck. Thus, there is a tradeoff between the length of an epoch and the overhead imposed on coordinators. So, we choose 10 ms to be the epoch length in Catenae in all the evaluations.

D. TPC-C

We implement TPC-C under the current specifications [27] with interfaces that propagate workload to Catenae, 2PL and OCC. TPC-C is an on-line transaction processing (OLTP) benchmark from the Transaction Processing Performance Council (TPC). Two representative operations in TPC-C benchmark, i.e. *NewOrder* and *OrderStatus*, are chosen and implemented for the evaluation.

1) Results: Figure 7 illustrates the evaluation results of Catenae, 2PL and OCC under extremely stressed TPC-C workload. The results are aggregated from the three operating DCs. Catenae is able to scale up from 25 clients/server to 100 clients/server under TPC-C *NewOrder* workload, after which its performance stays stable. 2PL and OCC follow similar scale up pattern. However, they only achieve roughly half of the throughput comparing to Catenae, which causes the doubling of latency. With more than 100 clients/server, there is a drop of throughput in OCC and 2PL because of high contention. The abort rate of 2PL increases when there are more conflicting transactions waiting for locks, since we have set a timeout on waiting for locks. OCC suffers from constantly significant abort rates under the *NewOrder* workload since there is extremely high read-write contention among transactions. Catenae maintains very low abort rates by efficiently scheduling concurrent transactions using transaction chain algorithm. It allows Catenae to achieve higher concurrency, which leads to a higher throughput of transaction execution. Additionally, the high success rate of speculative execution even under contended workload allows Catenae to commit transactions with low latency as shown in Figure 1. Thus, the faster transactions commit, the less contention is experienced in Catenae.

OrderStatus is a read-only transaction. Catenae judiciously processes read-only transactions in local DCs when the accessed records are not about to be committed to an updated

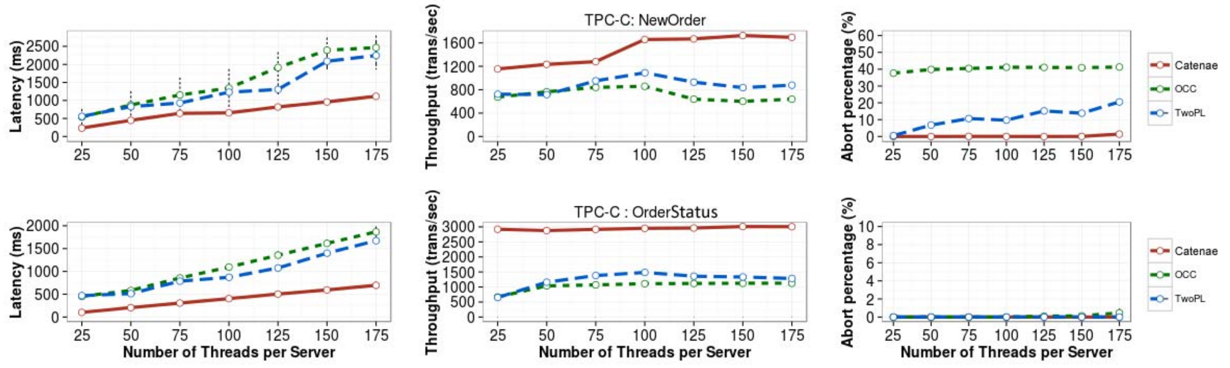

Fig. 7. Performance results of Catenae, 2PL and OCC under TPC-C NewOrder and OrderStatus workload

value. This condition is always true when running a read-only workload against Catenae. Thus, Catenae is able to commit read-only transactions locally, which significantly reduce the commit latency and boosts the throughput. In contrast, 2PL needs to check and obtain read locks across DCs while OCC requires to validate the read values across DCs. As a result, Catenae achieves more than twice the throughput of 2PL and OCC with nearly 70% less commit latency.

VIII. Performance and Liveness Tradeoff

A. Speculative Execution

Catenae provides efficient transaction support on top of fully replicated data stores, such as [28], [29], [17], [30], [31]. Since Catenae relies on a deterministic duration that a transaction accesses a specific data partition on a specific chain server, it is desirable, but not mandatory, to deploy chain servers symmetrically, i.e., using the same VM flavor to host the same namespace range, in all DCs. For performance predictability and cost control, it is common and reasonable to host the instances of a specific component of an application using the same VM flavors in today's Cloud platforms. Hosting the chain servers of Catenae asymmetrically among DCs will increase the possibility to have an inconsistent transaction execution dependencies during speculative executions among DCs. This will not influence the correctness of Catenae but triggering the catch up procedure and delaying the transaction commit to two RTTs, which is the same latency overhead comparing to the classic 2PL over Paxos commit. We validated in Section III-B, it is likely to achieve the same execution dependency in most of transactions speculatively executed using TC concurrency control. Speculative executions with consistent results in all DCs enable transactions to commit using a single RTT.

B. Liveness among Data Centers

Catenae does not pre-order transactions before execution, they are allowed to compete and concurrently execute at runtime. It maximizes the concurrency of transaction executions. However, Catenae expects DCs to execute the same set of transactions received from the epoch messages sent from all DCs, which leads to the most consistent transaction dependencies during speculative execution. The consistent dependencies during speculative execution allows Catenae to have extremely low commit latency, but comes with a tradeoff. An outage of a DC could cause other DCs to block waiting for its epoch message, which contains the transactions received in that DC. The blocking continues when the expected epoch messages are eventually delivered. This is similar to blocking scenarios in 2PL, that could be overcome by using state machine replication. Catenae applies a time-based delay tolerance technique to ascertain the state of a failed DC. Large delay tolerance may result in endless waiting for the epoch messages from a failed DC, that largely influences the performance. Small delay tolerance may neglect the transactions happened in the "suspected failed" DC and result in periodic high transaction abort rates in that DCs or a lot of transaction catch up workload across DCs. Thus, this design choice tradeoffs the high possibility to have consistent dependency during speculative execution with the complication of failure handling. Specifically, a configurable time-based offset is implemented in Catenae to record the differences between the epoch to execute a batch of transactions from all DCs, called the execution epoch (EE), and the local epoch counter in the coordinator, called the coordinator epoch (CE). The CEs are synchronized using the Network Timing Protocol (NTP). The EEs are deterministically calculated using the receipt epoch of transactions plus the estimated inter-DC transmission delay. If epoch messages are delayed during transmission, then the execution epoch of the delayed messages will be a past epoch in the recipient (coordinator). In this case, they are executed immediately if the time-based delay tolerance is not violated. In other words, the maximum offset between CE and EE is the delay tolerance, beyond which epoch messages are abandoned. The offset is expected to be dynamic during the operation of Catenae. The offset increases when the network among DCs introduces unexpected delays while delivering epoch messages. The offset decreases when the coordinators skip an execution epoch, which does not contain transaction distribution payload from all DCs. In this case, we guarantee that the CEs are always larger than EEs.

On the other hand, Catenae can be adapted to operate while receiving only majority epoch messages. Specifically, upon receiving the majority epoch messages, Catenae proceeds to transaction scheduling and execution phase. The incomplete receipt of transactions from DCs will lead to a higher possibility to have divergent transaction execution dependencies. The inconsistent execution dependency will be corrected by the selection of majority execution dependency during the validation phase with another RTT. Thus, the tradeoff allows Catenae to operate in a failure-prone environment but with a significant overhead for catching up inconsistent transaction executions. The possibility of having inconsistent transaction executions is evaluated in Section III-B and shown in Figure 1.

C. Liveness among Transactions

The transaction chain maintains serializability and liveness among transactions by ensuring that the dependencies among transactions are acyclic. A dependency is added to a pair of transaction by a chain server only when such dependency does not exist and will not generate cyclic dependency implicitly. Specifically, a chain server will not add a dependency contradictory to the dependency already embedded in the transaction. Dependency gradually propagates among chain servers with the passing of transactions and transactions are order linearly by chain servers with their observed dependency. Cyclic behavior can only happen when chain servers do not have enough information regarding some concurrent transactions, as shown in one example in Section VI-C. This kind of cyclic dependency is prevented by transaction resolver, who adds implicit dependency to transactions. Implicit dependencies are added when a superset of cyclic behaviors (as illustrated in Figure 4) are detected.

IX. Conclusion

We present Catenae, a geo-distributed transaction framework. It leverages novel epoch boundary synchronization protocol among DCs to improve transaction commit latency and extends the transactions chain algorithm to efficiently schedule and execute transactions in multiple DCs. We show that Catenae only needs one single inter-DC communication delay to execute non-conflicting geo-distributed read/write transactions and one RTT to execute potentially conflicting geo-distributed transactions most of the time. The worst case commit latency of Catenae requires two RTTs. Catenae achieves more than twice the throughput than 2PL and OCC with more than 50% less commit latency under TPC-C workload.

As a future work, we would like to investigate the performance of Catenae in a less ideal environment. Specifically, we would like to quantify the performance drop of Catenae using VMs that have heterogeneous specifications. The heterogeneity among VMs can emerge within or across data centers. Furthermore, the network among data centers also plays an essential role in guaranteeing the performance of Catenae. We would like to continue this work by evaluating and further improving its performance under severer network conditions, where epoch messages can be frequently delayed or lost.

Acknowledgment

This work was supported by the Erasmus Mundus Joint Doctorate in Distributed Computing funded by the EACEA of the European Commission under FPA 2012-0030 and the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research under the contract RIT10-0043.

References

[1] Summary of the amazon ec2 and amazon rds service disruption in the us east region. http://aws.amazon.com/message/65648/, 2011.

[2] Massive flooding damages several nyc data centers. http://www.datacenterknowledge.com/archives/2012/10/30/major-flooding-nyc-data-centers/, 2012.

[3] Summary of the december 24, 2012 amazon elb service event in the us-east region. http://aws.amazon.com/message/680587/, 2012.

[4] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[5] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

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

[7] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Lightweight multi-key transactions for key-value stores. *CoRR*, abs/1509.07815, 2015.

[8] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[9] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006.

[10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Ying Liu and V. Vlassov. Replication in distributed storage systems: State of the art, possible directions, and open issues. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2013 International Conference on*, pages 225–232, Oct 2013.

[12] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[13] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, Oakland, CA, May 2015. USENIX Association.

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

[15] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013.

[16] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[17] Ying Liu, Xiaxi Li, and V. Vlassov. Globlease: A globally consistent and elastic storage system using leases. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 701–709, Dec 2014.

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

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

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

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

[22] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 15–26, New York, NY, USA, 2014. ACM.

[23] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.

[24] Cloudping. http://www.cloudping.info/.

[25] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[26] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, June 1978.

[27] Tpc-c, the order-entry benchmark. http://www.tpc.org/tpcc/.

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

[29] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[30] Cosmin Arad, Tallat M. Shafaat, and Seif Haridi. Cats: A linearizable and self-organizing key-value store. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 37:1–37:2, New York, NY, USA, 2013. ACM.

[31] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.