Efficient Distributed Transaction Processing in Heterogeneous Networks

Qian Zhang[*]
Renmin University of China
zhangqiangzq@ruc.edu.cn

Jingyao Li[*]
Renmin University of China
li-jingyao@ruc.edu.cn

Hongyao Zhao[*]
Renmin University of China
hongyaozhao@ruc.edu.cn

Quanqing Xu
OceanBase, Ant Group
xuquanqing.xqq@oceanbase.com

Wei Lu[†]
Renmin University of China
lu-wei@ruc.edu.cn

Jinliang Xiao
OceanBase, Ant Group
xiaoshi.xjl@oceanbase.com

Fusheng Han
OceanBase, Ant Group
yanran.hfs@oceanbase.com

Chuanhui Yang
OceanBase, Ant Group
rizhao.ych@oceanbase.com

Xiaoyong Du[†]
Renmin University of China
duyong@ruc.edu.cn

ABSTRACT

Countrywide and worldwide business, like gaming and social networks, drives the popularity of inter-data-center transactions. To support inter-data-center transaction processing and data center fault tolerance simultaneously, existing protocols suffer from significant performance degradation due to high-latency and unstable networks. In this paper, we propose RedT, a novel distributed transaction processing protocol that works in heterogeneous networks. In detail, nodes within a data center are inter-connected via the RDMA-capable network and nodes across data centers are inter-connected via TCP/IP networks. RedT extends two-phase commit (2PC) by decomposing transactions into sub-transactions in terms of the data center granularity, and proposing a pre-write-log mechanism that is able to reduce the number of inter-data-center round-trips from a maximal of 6 to 2. Extensive evaluation against state-of-the-art protocols shows that RedT can achieve up to 1.57× higher throughputs and 0.56× lower latency.

PVLDB Reference Format:
Qian Zhang, Jingyao Li, Hongyao Zhao, Quanqing Xu, Wei Lu, Jinliang Xiao, Fusheng Han, Chuanhui Yang, and Xiaoyong Du. Efficient Distributed Transaction Processing in Heterogeneous Networks. PVLDB, 16(6): 1372 - 1385, 2023.
doi:10.14778/3583140.3583153

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dbiir/RedT/.

1 INTRODUCTION

Modern distributed database systems achieve scalability and availability by partitioning and replicating the data over multiple nodes. Each partition has one *primary replica* and multiple *secondary replicas*. The consensus among replicas is guaranteed by Paxos [29] or Raft [38]. The standard approach to processing a distributed transaction T is called *2PC-Paxos* that executes T through three phases: 1) execution phase, 2) prepare phase, and 3) commit phase. In the execution phase, the coordinator decomposes T into sub-transactions that are distributed to the corresponding primary replicas (a.k.a. participants). Once each primary replica completes local execution, the coordinator follows 2PC to commit/abort the sub-transactions. In the prepare phase, first, the coordinator notifies each primary replica to check whether the sub-transaction is ready to commit; then, every primary replica synchronizes writes in terms of redo logs to the secondary replicas if the sub-transaction is ready to commit, and sends a response to the coordinator after the synchronization; subsequently, after receiving responses from all primary replicas, the coordinator decides whether T commits or aborts. Before entering the commit phase, the coordinator synchronizes its decision to coordinator backups to ensure the availability of the decision in case of its failure. In the commit phase, the coordinator first notifies all primary replicas to commit/abort the sub-transactions, and each primary replica then synchronizes the decision to its secondary replicas. *2PC-Paxos* takes a maximum of 6 network round-trips, with 1 in the execution phase, 3 in the prepare phase, and 2 in the commit phase. Note that, some network round-trips can be neglected for read-only transactions.

There is an increasing trend to support inter-data-center (abbreviated as inter-DC) transaction processing and data center fault tolerance simultaneously. On the one hand, the wisdom to facilitate fast data delivery is geo-locality, i.e., storing clients' data in the nearby data center. Yet, the countrywide or worldwide business, e.g., e-commerce and gaming, requires to co-operate clients' data across data centers, and drives the popularity of inter-DC transactions, each of which reads/writes primary replicas located in multiple data centers. On the other hand, to achieve system availability that tolerates data center failures, replicas of the same partition are deployed across data centers. *Nevertheless, distributed transaction processing suffers from severe performance degradation due to high-latency and unstable networks.* Compared with the intra-data-center (abbreviated as intra-DC) network that typically takes 70us $\lesssim$ RTT $\lesssim$ 1ms, the inter-DC network has much higher latency, and is less stable.

[*]These authors contributed equally to this work.
[†]Wei Lu and Xiaoyong Du are the corresponding authors.

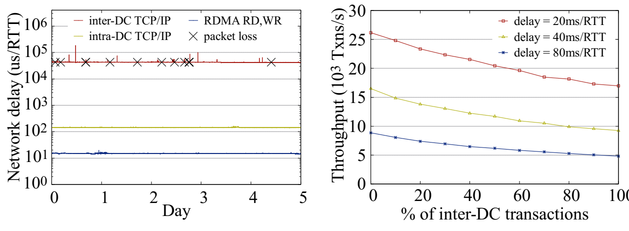
Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.
doi:10.14778/3583140.3583153

Figure 1(a) shows that the average round-trip time (abbreviated as RTT, a latency metric) from Beijing to Hong Kong ranges from a minimum of 40ms to a maximum of 183ms, and 13 packets are lost. We implement *2PC-Paxos* on distributed framework Deneva [23], in which each partition is set to have one primary replica and two secondary replicas. Replicas of the same partition are placed in different data centers. Figure 1(b) plots the system throughput over YCSB benchmark by varying the percentage of inter-DC transactions. The throughput decreases by a factor of 33%, 44%, and 45%, under the inter-DC latency of 20ms, 40ms, and 80ms, respectively.

To efficiently support inter-DC transaction processing and data center fault tolerance simultaneously, a few works attempt to reduce either the number or the overhead of network round-trips. TAPIR [63] and G-PAC [35] unite 2PC and consensus protocols in a single framework to eliminate redundant coordination, reducing the number of inter-DC round-trips from 6 to 3 in common cases. MDCC [28] incorporates Fast Paxos [30] protocol so that the number of round-trips is reduced from 6 to 2 under some prerequisites, e.g, in the absence of conflicts, while two additional round-trips are required otherwise. Multi-level 2PC [37] reduces the number of expensive inter-DC communications by layering participants, but cannot reduce the number of inter-DC round-trips. Because the inter-DC network could be a bottleneck, we aim to improve the system performance by further reducing either the number or the overhead of network round-trips.

Our research is motivated by the fact that in both academia and industry, the widespread use of RDMA-capable networks has become a reality. In academia, numerous works [6, 10, 14, 15, 54, 55, 58, 60] show that the performance of intra-DC distributed transaction processing can be improved by orders of magnitude over RDMA-capable networks. In industry, enterprises like Microsoft [21, 64], IBM [5] and Alibaba [9, 20] have reported boosting their database systems using RDMA-capable networks. Unfortunately, RDMA is built on lossless networks, and is currently considered only suitable for use in the local area networks (a.k.a. LAN) [64]. We realize heterogeneous networks are increasingly popular, where nodes within the same data center are inter-connected via RDMA networks, and nodes across multiple data centers are inter-connected via TCP/IP networks. Hence, redesigning distributed transaction processing in heterogeneous networks is of great necessity.

We propose RedT, an RDMA-enhanced distributed transaction processing protocol with availability guarantees. Compared with 2PC-Paxos, RedT reduces the number of inter-DC round-trips from a maximum of 6 to 2. Specifically, we design a pre-write-log mechanism. In this mechanism, during the execution phase, every primary/secondary replica is required to perform concurrency control, persist the redo logs, and send a commit/abort message to the coordinator. After collecting messages from at least $\frac{3}{4}$ replicas of each partition, the coordinator makes a commit/abort decision of the transaction, which is then propagated to every replica. After receiving the decision, every primary/secondary replica persists the commit/abort log, writes the data items if the transaction is committed, and releases the locks. RedT eliminates the synchronization of prepare messages from the coordinator to primary replicas, redo/commit/abort logs from the primary replicas to secondary replicas in the prepare/commit phase, and the transaction decision



(a) Latency microbenchmark

(b) Inter-DC transactions

Figure 1: Effect of the high-latency networks

among coordinator backups. Hence, RedT is able to reduce at most 4 inter-DC round-trips. Further, considering the network in which nodes within the same data center are inter-connected via RDMA networks, RedT selects one executor in each data center responsible for concurrency control over all replicas of the same data center. RedT achieves this by directly locking/unlocking/reading/writing replicas in other nodes using RDMA verbs. From this perspective, RedT extends sub-transactions from the replica granularity to the data center granularity, leading to a lower coordination cost and fewer inter-DC communications. Note that if we do not need to tolerate data center failures, for intra-DC transactions, RedT is reduced to a complete RDMA-boosted method. We provide a detailed analysis over the correctness of RedT to tolerate node failures and data center failures. We also perform a quantitative analysis over RedT and existing protocols to process different kinds of transactions, in terms of the inter-DC round-trips as well as communications, and show the advantages of RedT.

In summary, we make the following contributions:

- We present a novel distributed transaction processing protocol RedT that works in heterogeneous networks. It achieves high throughput and data center fault tolerance.
- We propose a pre-write-log mechanism that is able to reduce the number of inter-DC round-trips from a maximum of 6 to 2. Further, we employ the RDMA-capable network to relax the granularity from the replica to the data center, leading to a fewer number of inter-DC communications.
- We conduct extensive evaluations to compare RedT and four state-of-the-art protocols over YCSB and TPC-C benchmarks. The results show that RedT can achieve up to 1.57× higher throughputs and 0.56× lower latency.

2 PRELIMINARIES

2.1 Distributed Transaction Processing

As introduced in Section 1, the standard 2PC-Paxos takes a maximum of 6 network round-trips to process a distributed transaction. Take transaction T that consists of a write operation $w(x)$ and a read operation $r(y)$ in Figure 2 for an example. The primary replica of x/y locates in DC_1/DC_2, respectively. x/y has one secondary replica in DC_2/DC_1, and another secondary replica in DC_3/DC_3, respectively. T first takes one round-trip to access the primary replica of x and y in DC_1 and DC_2 (labeled as ① in Figure 2). After successful execution, T notifies the primary replica of x to commit (②), synchronizes redo logs to x's secondary replicas (③), and responds to the coordinator afterward. After collecting all responses

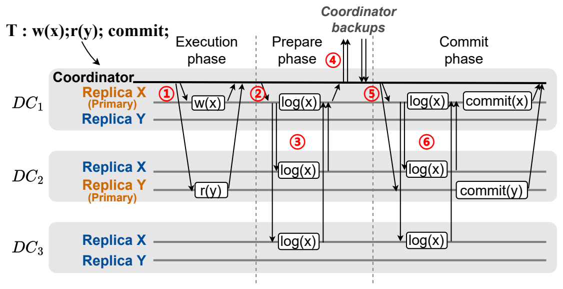


Figure 2: Modern distributed transaction processing

and before entering the commit phase, T synchronizes its decision to the coordinator backups for fault tolerance (④). In the commit phase, T notifies the primary replica of x and y to commit (⑤), and x synchronizes the decision to its secondary replicas (⑥).

2.2 RDMA

RDMA is a network feature that enables remote direct access to the main memory of a remote data node without the involvement of its CPU. Compared with traditional TCP/IP networks, RDMA can be built on InfiniBand to enjoy the advantages of high-bandwidth and low-latency. It also provides three properties to ensure high performance. (1) **Zero-copy** property. Applications can perform data transfers without the involvement of the network software stack. Data is sent and received directly to the buffers without being copied between the network layers. (2) **Kernel bypass** property. Applications can perform data transfers directly from user space without kernel involvement. (3) **No CPU involvement** property. Applications can access remote memory without consuming any CPU time in the remote machine. RDMA provides two categories of verbs for programming: (1) one-sided RDMA verbs, including READ, WRITE, WRITE With Immediate, and two atomic operations: FETCH-And-ADD (a.k.a. FAA) as well as COMPARE-And-SWAP (a.k.a. CAS), and (2) two-sided RDMA verbs, including SEND and RECEIVE. Programming using two-sided RDMA verbs can enjoy zero-copy and kernel-passing properties, and one-sided RDMA verbs can enjoy all three properties of RDMA. Note that, CAS and FAA are confined to operating data of 64-bit size per invocation. As a result, using a single CAS/FAA to operate data items and metadata together may exceed the limitation of 64-bit size.

2.3 RDMA-based Transaction Processing

RDMA-based transaction processing is based on the shared-memory architecture [15, 55, 58, 60]. As opposed to decomposing read/write operations of a transaction into sub-transactions in distributed databases, it follows the paradigm in the centralized databases to process transactions without the involvement of 2PC. In RDMA-based transaction processing, each node is equipped with multiple *executors*, responsible for executing transactions, and a *memstore*, a shared-buffer used to manage data items and redo logs for fault tolerance. Nodes are inter-connected via low-latency and high-bandwidth featured RDMA networks. To process a distributed transaction, each executor works like its counterpart in the centralized system and directly locks/unlocks/reads/writes data items stored on the remote nodes. Once all the read/write operations complete, the executor writes the redo logs into the remote memstore if necessary. The executor finally commits the transaction. Compared with traditional distributed transaction processing under TCP/IP networks, the performance of RDMA-based distributed transaction processing can be improved by a factor of at least 20× [15, 53].

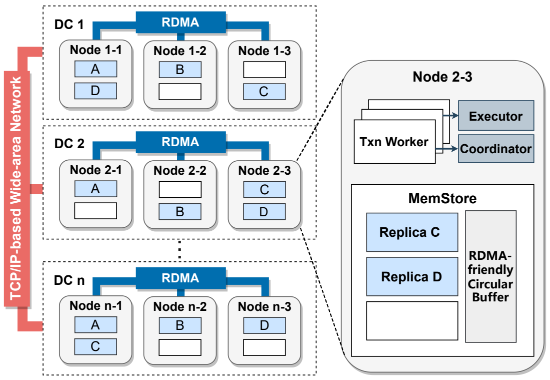


Figure 3: An overview of the system architecture

3 SYSTEM ARCHITECTURE

RedT is particularly designed for heterogeneous networks. An overview of the system architecture is shown in Figure 3. Each node is equipped with the following two kinds of components.

Multiple Txn Workers. A Txn Worker is able to act as either an **executor** or a **coordinator**. An executor executes sub-transactions by directly locking/unlocking/reading/writing data items that locate in other nodes of the same data center using RDMA verbs. A **coordinator** coordinates the execution of transactions.

MemStore. A MemStore is a pre-allocated RDMA-registered memory for maintaining replicas of different partitions. Besides, it maintains a circular buffer for logging. Any executors from other nodes in the same data center can directly write logs to this buffer via RDMA verbs. For performance, we assume MemStore is deployed on Non-Volatile Memory (NVM) by default, as many other works do [14, 27, 60, 62]. Otherwise, logs are persisted in the disk before committing for durability.

In RedT, a distributed transaction T_i is executed through two phases: 1) execution phase, and 2) commit phase. In the execution phase, we propose a *pre-write-log* mechanism, in which each primary/secondary replica involves performing concurrency control and writing the redo logs. Similar to 2PC-Paxos, the coordinator decomposes T_i into multiple sub-transactions in terms of the data center granularity. To simplify the discussion, we assume that a partition in each data center has at most one replica (later we will discuss how to handle the case without this assumption). A sub-transaction consists of the reads/writes from/to replicas that locate in the same data center. For example, in Figure 2, transaction T is decomposed into three sub-transactions, each of which consists of two operations $w(x)r(y)$ and locates in every data center. The *pre-write-log* mechanism eliminates the prepare phase that synchronizes the prepare messages from the coordinator to primary replicas, redo logs from the primary replicas to secondary replicas, and the transaction decision among coordinator backups.

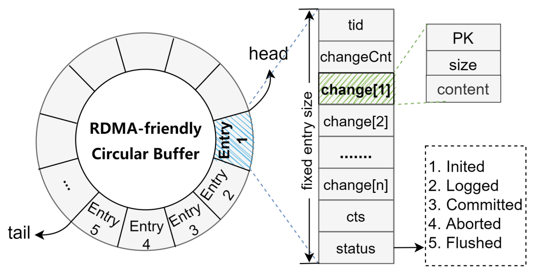


Figure 4: Data structure of the circular buffer

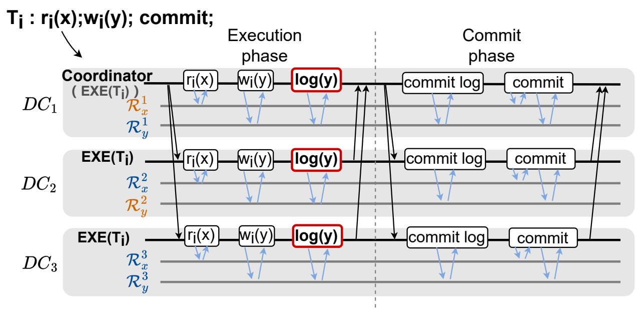


Figure 5: Inter-DC transaction processing

In each data center, one executor is selected to execute the sub-transaction, and follows the paradigm in the centralized databases to do concurrency control with a variant of two-phase locking protocol (a.k.a. 2PL), called No-Wait [7]. To be specific, the executor directly locks/reads/writes/unlocks data items that locate in other nodes of the same data center using RDMA verbs. Any conflicts of the locks on the same data items would cause the sub-transaction to abort. If the sub-transaction is ready to commit, the executor first writes the redo logs and then sends a commit message to the coordinator; otherwise, if the sub-transaction is ready to abort, the executor sends an abort message to the coordinator. In the commit phase, the coordinator collects the messages from each executor. Following the previous work [30], if more than $\frac{3}{4}$ of replicas per partition have the up-to-date data items that T_i reads/writes and vote to commit, the coordinator makes the commit decision, generates a commit timestamp, and notifies each executor to commit. Otherwise, the coordinator notifies each executor to abort. By doing this, RedT tolerates replica failures of no more than $\frac{1}{4}$ for any partition. We shall discuss the correctness of the number $\frac{3}{4}$ in Section 5.1.1. Note, for the sub-transaction votes to abort but the transaction decides to commit, the coordinator notifies its executor to re-write and commit the log. After receiving the message from the coordinator, each executor writes the data items if the coordinator decides to commit, commits/aborts the log, and releases the locks. We shall discuss the correctness in Section 4.2.2.

Discussion. We first discuss how to handle the case that multiple replicas of the same partition are deployed in the same data center, i.e., we do not tolerate data center failures. Let n be the number of replicas per partition in this data center. We logically split the replicas, that the transaction involves, into n groups, each of which takes one disjoint replica per partition. For each group in the data center, we select one executor that follows the same logic, as described before, to execute the sub-transaction in this group. We then discuss the difference between RedT and other protocols. In RedT, TAPIR [63] and G-PAC [35], the coordinator communicates with primary and secondary replicas, separately, to eliminate the synchronization of redo logs from the primary replicas to the secondary replicas. As opposed to that in TAPIR and G-PAC, every executor in RedT writes the redo logs directly to the replicas in the execution phase, further eliminating the prepare messages from the coordinator to the executors. For RedT and multi-level 2PC, they both reduce the number of inter-DC communications by layering the participants. As opposed to that in multi-level 2PC, RedT further reduces the number of participants within the same data center to a single participant, and transforms the distributed sub-transaction to the centralized sub-transaction using RDMA verbs.

4 TRANSACTION PROCESSING

In this section, we first introduce key data structures for concurrency control and logging. We then elaborate on how to do transaction processing. Finally, we analyze the number of inter-DC round-trips and communications.

4.1 Key Data Structures

To facilitate concurrency control, we propose an RDMA-friendly hash table that is used to efficient retrieval data items located in the other nodes of the same data center, as well as some auxiliary information. To facilitate logging, we propose an RDMA-friendly circular buffer that maintains redo logs only.

Data items and the auxiliary information. In each node, we maintain a set of primary/secondary replicas in its MemStore. For each data item x, x is associated with a latch $x.latch$, a lock $x.lock$, a timestamp $x.wts$, and a list $x.lockOwners$ of transactions that acquire the lock. $x.latch$ is used to prevent x from being modified simultaneously by other transactions. Note $x.lock$ is a compound data object with two variables $lockType$ and $lockNum$. The value of $x.lock.lockType$ is either $NULL$, or EX, or SH, meaning there is either no lock, or an exclusive lock, or one/more shared locks on x, respectively. $x.lock.lockNum$ represents how many shared locks on x are granted. To facilitate the implementation using RDMA verbs, the size of list $x.lockOwners$ is fixed. In this way, when the list is full, any transaction that applies to acquire the lock on x would abort. If the transaction commits or aborts, it will be removed from the list $x.lockOwners$. $x.wts$ is the maximum commit timestamp of transactions that have ever written x. It is used for the coordinator to check whether versions from different replicas of the same partition are consistent.

RDMA-friendly hash table. To make direct access to each data item x in any replicas from other nodes in the same data center using RDMA verbs, we follow [36] to build a special RDMA-friendly hash table where each key is the primary key, $x.PK$, of x, and its corresponding value is the address $x.addr$ of x in MemStore. Using $x.addr$, we can issue RDMA verbs to fetch x and its auxiliary information. Given a primary key $x.PK$, it is theoretically shown that, based on this hash table, it takes an average of 1.6 and a maximum of 3 RDMA invocations to obtain $x.addr$ [36].

RDMA-friendly circular buffer. We maintain a single RDMA-friendly circular buffer in the MemStore of each node. It stores redo logs only. An executor in one node can directly write log entries to the circular buffer of another node in the same data center using RDMA verbs. An overview of the circular buffer is shown in Figure 4. The buffer has a fixed number of slots, each of which can contain one log entry. In our design, a write is modeled as a change. a primary key $x.PK$ of data item x, *size* and *content* of the write. A log entry comprises *tid*, the ID of the transaction T, the number *changeCnt* of changes, the commit timestamp *cts* of T, and the status *status* of the transaction. To facilitate the writing of logs, we maintain two variables *head* and *tail* for each circular buffer. Because of the concurrent writes from different nodes, we carefully design the logging algorithm, which will be elaborated on in Section 4.2, to make sure the writes are safe.

Discussion. Note that, the circular buffer in our work is similar to that in Active-Memory [62]. Yet, we have two differences. On the one hand, Active-Memory uses undo log, while ours uses a redo log. On the other hand, we employ a single circular buffer per node, while Active-Memory employs a set of circular buffers per node with each for a separate target node to be communicated. For this reason, we can make better use of memory space, especially when the workloads to nodes are skewed. As the cost, concurrent updates to the buffer from multiple nodes could become an overhead. To solve this problem, we make a careful design of our logging algorithm, which will be discussed in Section 4.2.2.

4.2 Pre-Write-Log Mechanism

Thanks to the RDMA network, we present the pre-write-log mechanism, leveraging the shared-memory architecture to process sub-transactions in each data center. Next, we elaborate on how to do concurrency control and logging in each data center, respectively.

4.2.1 Concurrency Control. For brevity, given a data item x, let $\mathcal{R}_x^i$ be the i-th replica, $\mathcal{N}(\mathcal{R}_x^i)$ be the node that $\mathcal{R}_x^i$ locates in, and $\mathcal{N}(\mathcal{R}_x^i).DC$ be the data center that $\mathcal{N}(\mathcal{R}_x^i)$ locates in. In $\mathcal{N}(\mathcal{R}_x^i).DC$, we select one executor $EXE(T_i)$ to do concurrency control and logging. The main idea of No-Wait using RDMA is as follows.

In the execution phase, before any read $r_i(x)$ or write $w_i(x)$, $EXE(T_i)$ first issues an RDMA verb, namely RDMA_CAS, to acquire a *latch* on x from replica $(\mathcal{R}_x^i)$, which prevents x from being modified simultaneously by other transactions. If it fails to acquire the latch, T_i aborts; otherwise, $EXE(T_i)$ issues another RDMA verb, namely RDMA_READ, to read $x.lock$ which maintains all locks of x. If there exists a conflict with T_i on x, T_i aborts and releases the latch on x; otherwise, $EXE(T_i)$ acquires a new lock and releases the latch on x by issuing another RDMA verb, namely RDMA_WRITE in the remote node. Upon the sub-transaction aborts, the executor $EXE(T_i)$ replies the abort message to the coordinator. After the accomplishment of all operations, the executor $EXE(T_i)$ writes the redo logs, and replies the commit message to the coordinator.

In the commit phase, the coordinator collects the commit/abort messages from each executor $EXE(T_i)$. If more than $\frac{3}{4}$ replicas of each partition have the up-to-date data items that T_i reads/writes and vote to commit, the coordinator makes the commit decision, generates a commit timestamp, and notifies each executor to commit the sub-transaction; otherwise, it makes the abort decision and notifies each executor to abort the sub-transaction. The reason why we check how many replicas have the up-to-date data items is that due to the failure reasons, the minor portion of replicas may not be updated by the latest commit transaction, causing the later transactions to read the inconsistent data items. For each executor, after receiving the message from the coordinator, it writes the commit/abort log (discussed later), makes every write $w_i(x)$ to each replica once T_i is decided to commit, and releases all the granted locks if necessary.

Algorithm 1: RedT's data access functions

```
1  Function GetItemAddr($x.PK$):
2      return $x.addr \leftarrow hashTable(x.PK)$;
3  Function AcqLatch($x.addr$):
4      return $RDMA\_CAS(x.addr, x.latch, 0, 1)$;
5  Function AcqLock($x.PK, T_i, newLock$):
6      $x.addr \leftarrow GetItemAddr(x.PK)$;
7      while $\neg AcqLatch(x.addr)$ do goto line 7;
8      $x \leftarrow RDMA\_READ(x.addr)$;
9      if $conflict(x.lock, newLock)$ then
10         return $RDMA\_CAS(x.addr, x.latch, 1, 0)$;
11     $x.lock.lockType \leftarrow newLock.lockType$;
12     $x.lock.lockNum$ ++;
13     $x.lockOwners \leftarrow T_i.tid \cup x.lockOwners$;
14     $x.latch \leftarrow 0$; return $RDMA\_WRITE(x.addr, x)$;
15 Function RlsLock($x.PK, T_i, newValue$):
16     $x.addr \leftarrow GetItemAddr(x.PK)$;
17     while $\neg AcqLatch(x.addr)$ do goto line 16;
18     $x \leftarrow RDMA\_READ(x.addr)$;
19     $x.lock.lockNum$ - -;
20     $x.lockOwners \leftarrow x.lockOwners - T_i.tid$;
21     if $x.lock.lockNum = 0$ then $x.lock.lockType \leftarrow NULL$; ;
22     if $newValue \neq NULL$ and $x.wts < T_i.cts$ then
23         $x.data \leftarrow newValue$; $x.wts \leftarrow T_i.cts$;
24     $x.latch \leftarrow 0$; return $RDMA\_WRITE(x.addr, x)$;
25 Function Read($x.PK, T_i$):
26     $x.addr \leftarrow GetItemAddr(x.PK)$;
27     $newLock.lockType \leftarrow Sh$;
28     if $\neg AcqLock(x.PK, T_i, newLock)$ then $Abort$ $T_i$;
29     $x \leftarrow RDMA\_READ(x.addr)$;
30     $T_i.rs \leftarrow \{x\} \cup T_i.rs$;
31 Function Write($x.PK, newValue, T_i$):
32     $newLock.lockType \leftarrow Ex$;
33     if $\neg AcqLock(x.PK, T_i, newLock)$ then $Abort$ $T_i$;
34     $T_i.ws \leftarrow \{x\} \cup T_i.ws$;
35 Function Commit($T_i$):
36     foreach $k \in T_i.rs$ do $RlsLock(k.PK, T_i, NULL)$ ;
37     foreach $k \in T_i.ws$ do
38         $RlsLock(k.PK, T_i, newValue)$;
```

For illustration purposes, we list some key functions in Algorithm 1. Function *GetItemAddr* is used to obtain $x.addr$ of x by taking primary key $x.PK$ based on the RDMA-friendly hash table (lines 1–2). Function *AcqLatch* is used to acquire a latch on x to prevent x from being modified by other transactions simultaneously (lines 3–4). Function *AcqLock* is used to acquire an exclusive or shared lock on x. After acquiring the latch on x (line

7), $EXE(T_i)$ issues an *RDMA_READ* to obtain $x.lock$ and checks whether there is an conflict lock on x (lines 8–9). If there is an exclusive lock on x, or T_i attempts to acquire an exclusive lock while $x.lock.lockType = Sh$, it fails (line 10); otherwise, $EXE(T_i)$ updates $x.lock$ and $x.lockOwners$ to acquire a new lock and releases the latch by issuing *RDMA_WRITE* (lines 11–14). Following a reverse logic of function *AcqLock*, function *RlsLock* is used to release the remote lock and update the data item for write operations properly (lines 15–24). Note that we follow Thomas write rule [48] to update the data item (lines 22–23) with its correctness given in Section 4.2.2. Functions *Read* and *Write* are used to perform remote read and write. For read, $EXE(T_i)$ first tries to acquire a shared lock on x using function *AcqLock* (line 28). If the lock is acquired, $EXE(T_i)$ issues another *RDMA_READ* to read x and adds x to the read set $T_i.rs$ (lines 29–30). For write, $EXE(T_i)$ uses function *AcqLock* to acquire a remote exclusive lock on x, and adds x to the write set $T_i.ws$ if the lock is acquired (lines 33–34). Upon committing or aborting, $EXE(T_i)$ sequentially releases the locks (lines 35–38).

4.2.2 *Logging.*

Logging is performed in each data center to persist the outcome of concurrency control and the data written by sub-transactions. To do this, $EXE(T_i)$ first performs redo log replication to write redo logs to the log buffer in each data node containing corresponding replicas. Note that we ensure that the redo log replication is finished before entering the commit phase. After receiving the commit request from the coordinator, $EXE(T_i)$ then triggers commit log replication to update the status of T_i to committed/aborted. Finally, each data node has a log replay thread to update data items according to the persisted redo logs asynchronously.

Redo log replication. $EXE(T_i)$ performs the concurrency control over each replica separately. After the accomplishment of locking/reading/writing over replicas, $EXE(T_i)$ writes the redo logs to RDMA-friendly circular buffers in MemStores of the nodes that these replicas locate in using RDMA verbs. Take transaction T_i with two operations $r_i(x)$ and $w_i(y)$ in Figure 5 for an example. Because DC_1, DC_2, and DC_3 contain the replicas of x and y, we then select one executor in each data center to process the sub-transaction of T_i. During the execution phase, first, the coordinator packs $r_i(x)$ and $w_i(y)$ into the sub-transaction, and sends it to $EXE(T_i)$ for each data center; second, $EXE(T_i)$ performs the concurrency control over the replicas, and writes the redo logs only to the nodes with replicas to be written (highlighted in red of Figure 5), i.e., $EXE(T_i)$ in $DC_1/DC_2/DC_3$ writes the redo log of $w_i(y)$ to the circular buffer in $\mathcal{N}(\mathcal{R}_y^1)/\mathcal{N}(\mathcal{R}_y^2)/\ \mathcal{N}(\mathcal{R}_y^3)$ using RDMA verbs. If $EXE(T_i)$ fails to acquire the lock on a data item, it skips writing the redo logs. Finally, $EXE(T_i)$ notifies the commit/abort messages to the coordinator.

We present how to directly write redo logs to other data nodes of the data center using RDMA verbs in Algorithm 2. For each write $w_i(x)$ in a write set ws, $EXE(T_i)$ builds a change *chg* for x (line 11) using function *ConstructChange* (lines 1–3). $EXE(T_i)$ collects the changes into a batch that is sent to the same node (lines 10–12). For each node $\mathcal{N}$ in the data center, $EXE(T_i)$ builds a log entry *ent* based on the changes $chgs[\mathcal{N}]$, and writes *ent* directly to the circular buffer of $\mathcal{N}$ (lines 13–21). To do this, $EXE(T_i)$ reads the tail position of the target buffer by issuing an RDMA_FAA (line 15). If the tail position exceeds the head position by one round, existing log entries would be overwritten. To solve this problem, $EXE(T_i)$ issues an RDMA_READ to fetch the head position, and if an overwrite exists, $EXE(T_i)$ continuously issues RDMA_READs to fetch the head position until it is updated by the target node to make the tail position safely writable (lines 16–18). Note that, concurrent reads to the same buffer from different executors could happen. RDMA_FAA guarantees the atomicity that concurrent reads can return different tail positions. If the tail position is safely writable, $EXE(T_i)$ issues an RDMA_WRITE to write the log entry to the circular buffer (lines 19–21). By realizing that directly writing the log entry to the tail position without checking the head position is often safe, we make an optimization to reduce the number of RDMA_READs of head positions. we design a lazy check mechanism, in which $EXE(T_i)$ maintains a local head position *logHead* (lines 16–18). Only if the tail position exceeds *logHead* by one round, $EXE(T_i)$

Algorithm 2: RedT's log write functions

```
1  Function ConstructChange($x$):
2      $chg.PK \leftarrow x.PK$; $chg.size \leftarrow x.size$; $chg.content \leftarrow x.content$;
3      return $chg$;
4  Function ConstructEntry($chgs, tid, status$):
5      foreach $i \in [0, chgs.size)$ do $ent.change[i] \leftarrow chgs[i]$ ;
6      $ent.changeCnt \leftarrow chgs.size$;
7      $ent.tid \leftarrow tid$; $ent.cts \leftarrow 0$; $ent.status \leftarrow status$;
8      return $ent$;
9  Function WriteLog($T_i, ws, isActiveExe$):
10     foreach $x \in ws$ do
11         $chg \leftarrow ConstructChange(x)$;
12         $chgs[\mathcal{N}(\mathcal{R}_x^s)] \leftarrow chgs[\mathcal{N}(\mathcal{R}_x^s)] \cup chg$;
13     foreach node $\mathcal{N}$ and $\neg chgs[\mathcal{N}].empty()$ do
14         $ent \leftarrow ConstructEntry(chgs[\mathcal{N}], T_i.tid, Logged)$;
15         $idx \leftarrow RDMA\_FAA(\mathcal{N}.logBuf.tail.addr, 1)$;
16         while $idx - logHead[\mathcal{N}] \geq BUFSIZE$ do
17             $head\_addr \leftarrow \mathcal{N}.logBuf.head.addr$;
18             $logHead[\mathcal{N}] \leftarrow RDMA\_READ(head\_addr)$;
19         $ent\_addr \leftarrow \mathcal{N}.logBuf.entry[idx \ \% \ BUFSIZE].addr$;
20         $RDMA\_WRITE(ent\_addr, ent)$;
21         $T_i.logSet \leftarrow T_i.logSet \cup \langle ent\_addr, ent \rangle$;
22 Function CommitLog($T_i$):
23     foreach $\langle ent\_addr, ent \rangle \in T_i.logSet$ do
24         $ent.cts \leftarrow T_i.cts$; $ent.status \leftarrow T_i.status$;
25         $RDMA\_WRITE(ent\_addr, ent)$;
26 Function ReplayLog($\mathcal{N}$):
27     $move \leftarrow true$;    $h \leftarrow \mathcal{N}.logBuf.head$;
28     $t \leftarrow min(\mathcal{N}.logBuf.tail, h + BUFSIZE)$;
29     foreach $idx \in [h, t)$ do
30         $ent \leftarrow \mathcal{N}.logBuf.entry[idx \ \% \ BUFSIZE]$;
31         if $ent.status = Committed$ then
32             Apply the changes of $ent$ in node $\mathcal{N}$ locally;
33         if $ent.status = Committed$ or $Aborted$ or $Flushed$ then
34             $ent.status \leftarrow Flushed$;
35             if $move$ then
36                 $ent.status \leftarrow Inited$; $++\mathcal{N}.logBuf.head$;
37         else $move \leftarrow false$ ;
```

Table 1: A comparison of existing protocols with RedT in terms of inter-DC communications and round-trips. T_{ne} is a non-read-only inter-DC transaction, T_{re} is a read-only inter-DC transaction, T_{na} is a non-read-only intra-DC transaction, and T_{ra} is a read-only intra-DC transaction. $K = \sum_{i=2}^{n}(a_i + b_i)$.

Protocol	Phase	# of communications	# of round-trips			
			T_{ne}	T_{re}	T_{na}	T_{ra}
2PC-Paxos	exe	$2K$	1	1	0	0
	prep	$2\sum_{i=2}^{n} a_i+2m\sum_{i=1}^{n} a_i+2M$	2~3	1	2	1
	commit	$2K + 2m\sum_{i=1}^{n} a_i$	2	1	1	0
EP-Paxos	exe	$2K + 2m\sum_{i=1}^{n} a_i + 2M$	3	2	2	1
	commit	$2K + 2m\sum_{i=1}^{n} a_i$	2	1	1	0
MDCC	commit	$2\sum_{i=2}^{n}(a_i + b_i + c_i + d_i)$	1	1	1	1
		$2K + 2m\sum_{i=1}^{n}(a_i + b_i)$	2	2	1	1
TAPIR /G-PAC	exe	$2\sum_{i=2}^{n}(a_i + b_i + c_i + d_i)$	1	1	1	1
	prep	$2\sum_{i=2}^{n}(a_i + c_i)$	1	0	1	0
	commit	$2\sum_{i=2}^{n}(a_i + b_i + c_i + d_i)$	1	1	1	1
RedT	exe	$2(n-1)$	1	1	1	1
	commit	$2(n-1)$	1	1	1	1

then issues an RDMA_READ to read the head position and update *logHead* properly (lines 17–18). Due to a later update of logs in the commit phase, $EXE(T_i)$ locally maintains all memory addresses of log entries in other nodes (lines 21).

Commit log replication. After receiving the message from the coordinator, each executor enters the commit phase. We first discuss the case when the coordinator decides to commit. If $EXE(T_i)$ votes to commit, it writes the commit log by updating cts and $status$ of the redo log to $T_i.cts$ and *Committed*; otherwise, $EXE(T_i)$ modifies the status of the sub-transaction from aborted to committed, and follows the logic of function *WriteLog* to write a redo log by setting $cts = T_i.cts$ and $status = Committed$. We now discuss why it is correct to directly modify the status of the sub-transaction. Because the commit decision of T_i indicates that T_i still holds locks on more than $\frac{3}{4}$ of replicas for each partition, and we follow 2PL to do concurrency control, for any concurrent transaction T_j with conflicts with T_i, the serializable order between T_i and T_j is either $T_j \rightarrow T_i$ if T_j has already committed or otherwise $T_i \rightarrow T_j$. To help write the data items in less than $\frac{1}{4}$ of replicas that vote to abort, we write the redo log to the circular buffers in the corresponding data nodes and replay the logs asynchronously after the commit of T_i. If ($T_j \rightarrow T_i$) or ($T_i \rightarrow T_j$ and T_j has not committed), replaying the log does not affect the correctness of T_j; If $T_i \rightarrow T_j$ and T_j have committed, replaying the logs would overwrite the data items that T_j writes. To solve this problem, we follow Thomas write rule to apply all changes of T_i. We then discuss the case when the coordinator decides to abort. If $EXE(T_i)$ votes to commit, it writes the abort log by setting $status$ of the redo log to *Aborted*; otherwise, it returns. Details of commit log replication in each executor are given in function *CommitLog* of Algorithm 2, which is self-explained.

Asynchronous redo log replay. Because data items in replicas might not be updated even after the transaction commits due to the node failures, to make them consistent with the other replicas, we create a separate thread in each node that continuously scans the circular buffer and replays redo logs for replicas if its status is committed. In the *ReplayLog* function of Algorithm 2, we scan all entries from head to tail in the local log buffer (lines 27–29). Once encountering an entry with a $Committed$ status, we follow Thomas write rule to apply all changes (lines 31–32). That is, a change is replayed on a data item x only when the commit timestamp cts recorded in the entry is larger than $x.wts$, where $x.wts$ is cts of the last entry or transaction that modifies x. Finally, we update the status of each entry with $Aborted$ or $Committed$ to $Flushed$, and if the head is movable and the entry of the head position can be released for use, we set the status of the entry to $Inited$ and move forward of the head by a slot (lines 33–36). The head cannot move forward any further during this round of scan once encountering an entry with the $Logged$ or $Inited$ status (line 37).

4.3 Analysis on Inter-DC Round-trips

Given a non-read-only inter-DC transaction T_{ne}, let $\mathcal{DC}(T_{ne})$ be the collection of data centers that T_{ne} involves. For brevity, symbol n is set to the number of data centers in $\mathcal{DC}(T_{ne})$, i.e., $n = |\mathcal{DC}(T_{ne})|$, m ($m > 0$) is the fixed number of secondary replicas per partition, and M ($M > 0$) is the fixed number of backups for each coordinator. In each $DC_i \in \mathcal{DC}(T_{ne})$, suppose that T_{ne} writes/reads data items to/from primary (secondary) replicas in a collection of a_i (c_i) / b_i (d_i) nodes , respectively. That is, $a_i = |\{\mathcal{N}(\mathcal{R}_x^p)|x \in T_{ne}.ws\}|$, $b_i = |\{\mathcal{N}(\mathcal{R}_x^p)|x \in T_{ne}.rs\}|$, $c_i = |\{\mathcal{N}(\mathcal{R}_x^s)|x \in T_{ne}.ws\}|$ and $d_i = |\{\mathcal{N}(\mathcal{R}_x^s)|x \in T_{ne}.rs\}|$. By default, if there are overlaps, nodes are counted only once with the priority of a_i, b_i, c_i, and d_i in descending order. For illustration, let $K = \sum_{i=2}^{n}(a_i + b_i)$. Table 1 summarizes the number of inter-DC communications and inter-DC round-trips to process T_{ne} for RedT and other state-of-the-art protocols. To be fair, all protocols are analyzed under serializability and use 2PL for concurrency control. Without loss of generality, the coordinator of T_{ne} is assumed to locate in $DC_1 \in \mathcal{DC}(T_{ne})$. Compared with that of inter-DC communications over TCP/IP network, the cost of intra-DC communications using RDMA-capable networks is negligible (see Figure 1(a)). Thus, we only take the former into account.

For 2PC-Paxos, there are 5 to 6 inter-DC round-trips in total to process T_{ne}. We now step through the processing with reference to Figure 2. For round-trip ① in the execution (abbreviated as exe) phase, $2K$ communications are required for the coordinator to send/receive messages to/from the primary replicas. The prepare (abbreviated as prep) phase takes a maximum of 3 round-trips including ②, ③ and ④, for which $2\sum_{i=2}^{n} a_i$, $2m\sum_{i=1}^{n} a_i$ and $2M$ communications are required, respectively. For the special case that all primary replicas in $T_{ne}.ws$ locate in DC_1, i.e., $\sum_{i=2}^{n} a_i = 0$, round-trip ② can be eliminated, resulting in only 2 inter-DC round-trips in the prepare phase. Similarly, in the commit phase, processing T_{ne} requires 2 inter-DC round-trips that are ⑤ and ⑥, and $2K+2m\sum_{i=1}^{n} a_i$ communications in total, where $2K$ is the number of communications that are used to release the locks on data items in other nodes. As an optimization of the standard 2PC-Paxos protocol, Early Prepare [46] reduces one potential round-trip by combining the prepare phase with the execution phase. Early Prepare also writes redo logs in the execution phase, as RedT does. However, unlike Early Prepare, which runs sub-transactions in data node granularity, RedT runs sub-transactions in data center granularity, and therefore achieves lower coordination overhead and fewer inter-DC communications, which is verified in Table 2. Besides, Early Prepare is proposed for systems without replication, while RedT is designed for replicated systems. To make a fair comparison, we extend it to the replicated system by introducing the standard Paxos protocol for replication. We name this implementation EP-Paxos for brevity, which requires 5 inter-DC round-trips in total to process T_{ne}. As opposed to 2PC-Paxos, MDCC [28] tries to commit T_{ne} after the execution phase. In the absence of conflict or under the assumption of commutative operations, MDCC can commit with a single round-trip and $2\sum_{i=2}^{n}(a_i + b_i + c_i + d_i)$ communications by accessing all replicas directly. To be fair, we do not consider the commutative optimization, and set MDCC under serializability. In the case that any conflict exists, processing T_{ne} using MDCC requires 2 additional network round-trips and $2K+2m\sum_{i=1}^{n}(a_i+b_i)$ communications, which are similar to that in the commit phase of standard 2PC-Paxos. TAPIR [63] or G-PAC [35] requires 3 inter-DC round-trips to process T_{ne}. In the execution phase, the coordinator sends messages to all replicas involved in a single round-trip. In the prepare phase, the coordinator directly replicates the redo logs to the primary and secondary replicas. As a result, it requires 1 round-trip and at least $2\sum_{i=2}^{n}(a_i + c_i)$ communications. Similar to the execution phase, in the commit phase, the coordinator needs $2\sum_{i=2}^{n}(a_i + b_i + c_i + d_i)$ communications with all replicas accessed in other data centers in a single round-trip. RedT requires exactly 2 inter-DC round-trips. In the execution phase, the coordinator sends/receives messages to/from $EXE(T_{ne})$ in other $n-1$ data centers in 1 round-trip. The coordinator then decides the committed/aborted status of T_{ne} locally by collecting the votes from the messages. In the commit phase, the decision is sent to each $EXE(T_{ne})$ in other $n-1$ data centers. After the decision is synchronized, each $EXE(T_{ne})$ notifies the coordinator with a total of $n-1$ communications.

We extend the analysis from T_{ne} to other three kinds of transactions T_{re}, T_{na} and T_{ra} in Table 1. For read-only transactions T_{re} and T_{ra}, we have $a_i = 0$ and $c_i = 0$, for $1 \le i \le n$, meaning that it is unnecessary to synchronize redo logs among writes. For intra-DC transactions T_{na} and T_{ra}, we have $a_i = 0$ and $b_i = 0$, for $2 \le i \le n$, meaning that $K = 0$ and it is unnecessary for the coordinator to send messages to primary replicas in other data centers. For reference, we list the number of inter-DC round-trips for T_{re}, T_{na} and T_{ra} in the last three columns of Table 1, respectively. To summarize, in most cases, RedT enjoys a lower or at least equal number of inter-DC round-trips and inter-DC communications compared with existing protocols.

5 FAULT TOLERANCE AND CORRECTNESS

5.1 Fault Tolerance

5.1.1 Node Fault Tolerance. In our design, a node can act as either a data node, an executor, or a coordinator. Therefore, when a node fails, we need to perform failure recovery for a loss of data nodes and coordinators/executors.

Failure recovery of data nodes. As opposed to Paxos/Raft, RedT follows a stricter condition in which as long as more than $\frac{3}{4}$ of the replicas for each partition are still alive, the transaction keeps running normally; otherwise, if less than $\frac{3}{4}$ of the replicas for any partition vote to commit within a pre-defined timeout, the coordinator directly enters the commit phase and notifies the replicas to abort. Upon a failure, it is unnecessary for the replicas in the failed nodes to release the locks. Instead, RedT finds new data nodes, restores the failed replicas, and synchronizes the logs from the other replicas to the new replicas following the logic of Paxos.

Failure recovery of executors. In each data center, there is an executor that is responsible for acquiring/releasing locks on all replicas operated by T in the same data center, and these locks are directly maintained with the replicas. In this way, the executor is stateless. For this reason, once the executor fails, the coordinator chooses another executor in the same data center to replace the failed executor and lets it manage the locks on the replicas.

Failure recovery of coordinators. In RedT, the coordinator maintains the operations and current status of the transaction T and is responsible for coordinating the execution of T. In the execution phase, the coordinator sends the operations of T to all executors. Once the coordinator fails, one executor is selected to act as the new coordinator. Because the status of T located in the failed coordinator is lost, we need to restore the status in the new coordinator. To do this, we adapt the mechanism used in TAPIR [63]. The new coordinator restores the status by collecting the status of every healthy replica. If more than $\frac{1}{2}$ of the replicas for each partition are still alive and have the same commit decisions, it means that the status is committed; otherwise, it means that the status is aborted. The reason is as follows. The status of a transaction is committed only if more than $\frac{3}{4}$ of replicas for each partition vote to commit, meaning that at the worst case, $\frac{1}{4}$ of replicas vote to commit fail. In this way, $\frac{1}{2}$ ($\frac{3}{4} - \frac{1}{4} = \frac{1}{2}$) of replicas still vote to commit. Thus, by collecting the decisions from at least $\frac{1}{2}$ of replicas for each partition, the new coordinator can restore the status of the transaction. If more than $\frac{1}{4}$ of the replicas fail, the new coordinator needs to wait until partial replicas are recovered.

5.1.2 Data Center Fault Tolerance. When a minor portion of data centers fail, in the worst case, the coordinator and some executors of T, and some replicas accessed by T are lost. In this way, among the remaining healthy executors, one executor is selected to act as the new coordinator following the same logic discussed in Section 5.1.1. If the number of failed replicas does not exceed $\frac{1}{4}$ of replicas for each partition, as discussed in Section 5.1.1, the new coordinator does not need to restore the executor and replicas of T; otherwise, when multiple data centers fail, causing more than $\frac{1}{4}$ of replicas for some partitions lost, the new coordinator needs to wait until the replicas recover. To keep the system having enough healthy replicas, RedT follows the logic of Paxos to find new data nodes and restore the failed replicas asynchronously.

5.2 Correctness

To prove correctness, we show that RedT maintains the following three properties 1) isolation, 2) atomicity, and 3) durability given up to $\frac{1}{4}$ of replicas failure in each partition.

Isolation. In RedT, any two conflicting transaction, T_i and T_j, that violate the serializability cannot commit both. That is, T_i and T_j access the same data item x, and satisfy the order $T_i \to T_j$ and $T_j \to \dots \to T_i$. As discussed, in RedT, a transaction can commit only after acquiring locks on more than $\frac{3}{4}$ of replicas for each partition.

T_i and T_j cannot both acquire locks on more than $\frac{3}{4}$ of replicas. Therefore, with the order $T_i \rightarrow T_j$, T_j can acquire the lock on more than $\frac{3}{4}$ of replicas only after T_i commits and releases its locks. By using No-Wait, the order $T_j \rightarrow \ldots \rightarrow T_i$ cannot be held because T_i cannot acquire other locks after releasing the locks. Even if less than $\frac{1}{4}$ of the replicas fail, more than $\frac{1}{2}$ of the replicas still maintain the correct orders. Because we can compute the transaction status based on the replicas, failures of the coordinator and the executors do not affect the serializable scheduling of transactions.

Atomicity. For atomicity, we must guarantee that T is finally committed if T enters the commit phase. Barring failures, the coordinator would commit T normally; otherwise, the remaining executor restores a new coordinator, and the new coordinator also computes the same transaction status following the logic in Section 5.1 and then commits T normally.

Durability. For any committed transaction T, upon a failure, the original/new coordinators obtain the same status of T and let executors do the same writes to the replicas. Note, for the replicas that vote to abort and the coordinator decides to commit, the executor modifies the status of the sub-transaction and writes a redo log with a commit timestamp. Then the asynchronous thread replays the redo logs according to the order of the commit timestamp, thus maintaining the original serializable orders.

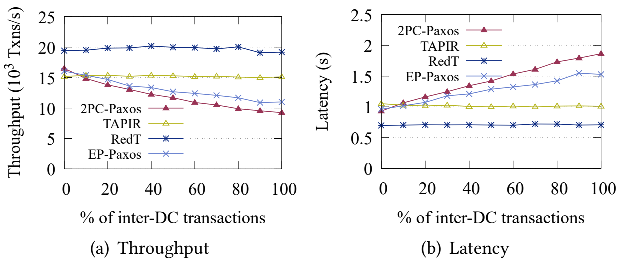


Figure 6: Impact of inter-DC transaction ratio - YCSB

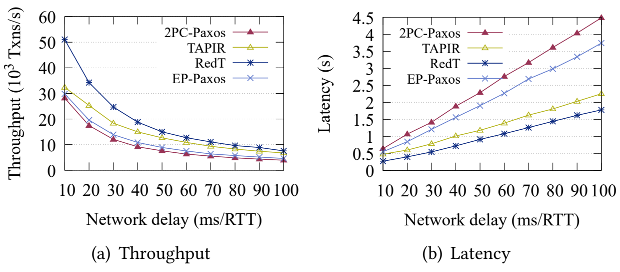


Figure 7: Impact of inter-DC network delay - YCSB

6 EXPERIMENT

We conduct the experiments over 8 or 12 machines of an RDMA-capable EDR cluster. Each machine is equipped with one Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz (18 cores×2 HT) processor, 128GB RAM, and one ConnectX-5 EDR 100Gb/s InfiniBand MT27800. All experiments are conducted over the following two benchmarks.

YCSB [11] is a comprehensive benchmark that simulates large-scale Internet applications. Its dataset contains a single 10-column relation, in which each data item occupies 1KB. The relation is then horizontally partitioned, and partitions are distributed to data nodes in a round-robin manner. In our evaluation, each data node maintains 4 million data items, resulting in a 4GB storage space per data node. Each transaction of the workloads is set to have a fixed number of 10 read/write operations that access data items following the Zipfian distribution. A larger Zipfian value results in a higher contention workload. **Unless otherwise specified, we set the skew factor to 0.2, and the write-ratio to 0.5**, i.e., there are 50% reads and 50% writes among all transactions.

TPC-C [50] is another popular OLTP benchmark simulating a warehouse order processing application. Its dataset is composed of 9 relations, in which each warehouse has 100MB of data. By default, we set the number of warehouses per node to 32. TPC-C contains 5 types of transactions, among which Payment, New-order, and Delivery are read-write transactions, Stock-level and Order-status are read-only transactions. Following the previous works [23, 59], we only evaluate Payment and New-order transactions in the experiments and omit Delivery, Stock-level, and Order-status that are local transactions.

By default, we set up 4 data centers, with 2 nodes in each. To support the data center fault tolerance, we set the replication factor to 3, and replicas of each partition are placed in 3 distinct data centers. As we mainly focus on the processing of inter-DC transactions, unless otherwise specified, the ratio of inter-DC transactions is set to 1 in YCSB and 0.5 in TPC-C. Each transaction is confined to 2 nodes. We simulate the inter-DC round-trip network delay as 40ms, which is reasonable. For example, RTT from New York to Dallas is 40ms [31]. We compare RedT with four protocols 2PC-Paxos, TAPIR, EP-Paxos (Early Prepare with Paxos), and MDCC. In addition, we also implement RedT without RDMA networks. To avoid an apple-to-orange comparison, all protocols are implemented and evaluated in Deneva [23], open-sourced by MIT. For reference, we release the source code of implementations via [43].

6.1 Impact of Inter-DC Transaction Ratio

We first evaluate the performance by varying the percentage of inter-DC transactions in Figure 6. We can observe that RedT performs the best, followed by TAPIR, EP-Paxos, and 2PC-Paxos. As expected, compared to 2PC-Paxos, EP-Paxos exhibits a higher throughput and lower latency given the same ratio of the inter-DC transactions, by eliminating the barrier between the execution phase and prepare phase. The performance of both 2PC-Paxos and EP-Paxos drops linearly when the ratio increases. This is because inter-DC transaction processing takes a constantly larger number of inter-DC round-trips than intra-DC transaction processing. In contrast, RedT and TAPIR perform rather stable when the ratio increases. This is because, for either RedT or TAPIR, it takes the same fixed number of inter-DC round-trips to process inter-DC or intra-DC transactions. RedT outperforms TAPIR because it requires a less or equal number of round-trips to process distributed transactions.

6.2 Impact of Inter-DC Networks

We use the traffic control emulator *tc* provided by Linux kernel to simulate a fixed and jittered inter-DC RTT ranges from 10ms to 100ms in Section 6.2.1, and Section 6.2.2, respectively.

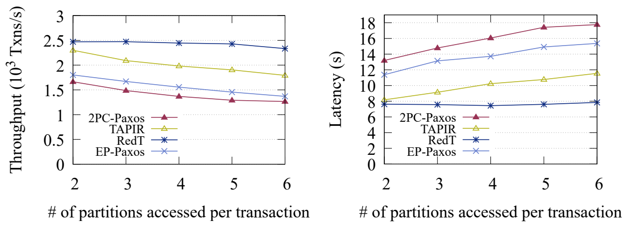
Table 2: An experimental comparison of all protocols in terms of inter-DC communications and round-trips to process non-read-only inter-DC committed transactions.

Protocol	Phase	communications		round-trips	
		average #	max #	average #	max #
2PC-Paxos	exe	2.00	2	1	1
	prep	13.49	14	2.97	3
	commit	9.60	10	2	2
EP-Paxos	exe	13.59	14	3	3
	commit	9.60	10	2	2
MDCC †	commit	7.43 (9.24)	20 (20)	1.02 (1.38)	3 (3)
TAPIR /G-PAC	exe	7.33	10	1	1
	prep	7.01	10	1	1
	commit	7.34	10	1	1
RedT	exe	5.87	6	1	1
	commit	5.87	6	1	1

† For MDCC, we also report the results under high contention (i.e., the skew factor is set to 0.8) in parenthesis. For all other protocols, the skew factor is set to 0.2 by default.

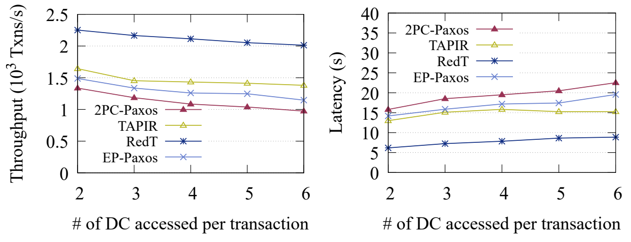
6.2.1 Impact of Inter-DC Network Delay. In order to show the adverse effect of the high-latency inter-DC networks, we report the performance under various inter-DC network delays for inter-DC transactions. Figure 7 demonstrates the throughput and latency over YCSB. As the system deployed across multiple data centers is exposed to and bottlenecked by the inter-DC networks, its performance drops sharply with a growing inter-DC network delay. Take 2PC-Paxos for an example. Its throughput drops by 87% with the network delay increasing from 10ms to 100ms. Benefitting from its less exposure to inter-DC networks with a smaller number of inter-DC round-trips and communications, as expected, RedT excels 2PC-Paxos, EP-Paxos, and TAPIR in terms of throughput and latency under any given network delay.

6.2.2 Impact of Inter-DC Network Communications. We report the number of inter-DC network communications to process non-read-only inter-DC committed transactions in Table 2. Due to the space limitation, we omit the experiments over the other kinds of transactions, and instead, we make a theoretical analysis in Table 1. Consistent with the theoretical analysis in Table 1, RedT enjoys the smallest number of communications, with the average/maximum number per transaction being 11.74/12, while for 2PC-Paxos, EP-Paxos, and TAPIR, these numbers are 25.09/26, 23.19/24, and 21.68/30, respectively. We also report these numbers for MDCC, which are discussed later in Section 6.4. To evaluate the benefit of this reduction of communications, we plot the throughput and latency of all protocols by varying the partitions accessed per transaction in Figure 8. In this experiment, 12 machines are used to form 4 data centers, and primary replicas accessed by each transaction are confined to 2 data centers. With the number of partitions accessed per transaction increasing from 2 to 6, all protocols except RedT are exposed to a growing number of inter-DC communications. The throughput of 2PC-Paxos, EP-Paxos, and TAPIR, drops by 27%, 23% and 15%, and their latency increases by 42%, 37% and 17%, respectively, while both the throughput and latency of RedT



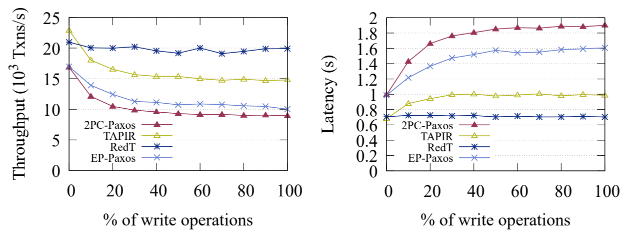
(a) Throughput (b) Latency

Figure 8: Impact of partitions accessed - YCSB



(a) Throughput (b) Latency

Figure 9: Impact of data centers accessed- YCSB



(a) Throughput (b) Latency

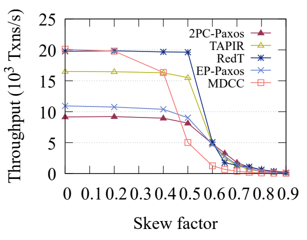
Figure 10: Impact of write-ratio - YCSB

stay stable in any cases. We also vary the number of data centers involved per transaction, and report the results in Figure 9. The performance of all protocols decreases because of an increasing number of communications. However, RedT is less exposed to the unstable inter-DC network by decomposing transactions in terms of data center granularity. As a result, the performance of 2PC-Paxos, EP-Paxos, and TAPIR drops when the number of data centers involved per transaction varies from 2 to 6, while that of RedT decreases less.

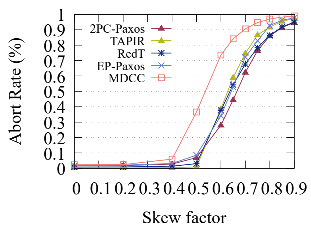
6.3 Impact of Read-Write Ratio

We evaluate the performance by varying the write-ratio over YCSB. Figure 10 shows the throughput and latency of different protocols, with the write-ratio ranging from 0% to 100% among all inter-DC transactions. For read-only transactions, i.e., setting the write-ratio to 0%, 2PC-Paxos and EP-Paxos (TAPIR and RedT) demonstrate similar performance. This is because, for all protocols, it is not necessary to synchronize the redo/commit logs to secondary replicas in the write set. In this way, 2PC-Paxos and EP-Paxos (TAPIR and RedT) take three (two) round-trips and coincide with each other.

However, as the write-ratio increases, the performances of these protocols begin to diverge. For 2PC-Paxos, its performance drops

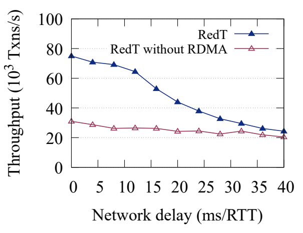


(a) Throughput

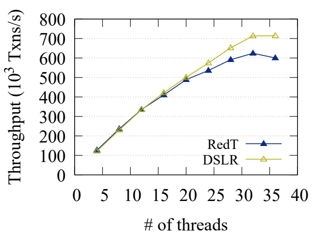


(b) Abort rate

Figure 11: Impact of contention level - YCSB

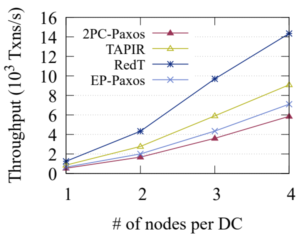


(a) Performance with/without RDMA

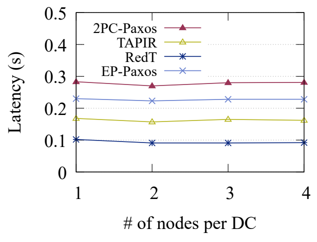


(b) Comparison of RDMA protocols

Figure 12: Impact of RDMA - YCSB

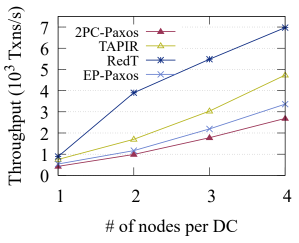


(a) Throughput

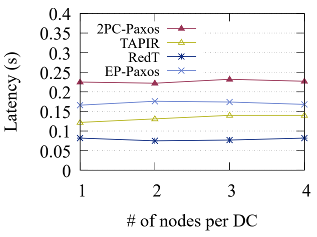


(b) Latency

Figure 13: Scalability - YCSB



(a) Throughput



(b) Latency

Figure 14: Scalability - TPC-C (New-order)

sharply. For example, the latency of write-only transactions is approximately 1.92× higher than that of read-only transactions. This is because each non-read-only inter-DC transaction takes five to six round-trips, as demonstrated in Table 1. Often, a moderate write-ratio would result in a vast number of non-read-only transactions. For example, when the write-ratio is set to 0.5, 99.9% transactions would become non-read-only transactions. Similarly, the latency of EP-Paxos/TAPIR increases by 1.62%/1.44% as the number of round-trips required increases from two/three (for read-only transactions) to three/five (for write-only transactions). In RedT, all kinds of transactions constantly take 2 inter-DC round-trips. Although extra logging and data updating overheads are introduced for non-read-only transactions, these overheads come imperceptible compared with the cost of inter-DC round-trips, resulting in RedT's insensitivity to the changing write-ratio.

6.4 Impact of Contention Level

We evaluate the performance by varying the contention level over YCSB, and show the results in Figure 11. Under low contention levels (i.e., skew factor < 0.4), as we can see, all protocols have a low abort rate and perform rather stable. In this situation, the number of inter-DC round-trips almost dominates the performance. RedT and MDCC take the minimum number of round-trips, and thus outperform the others. Under moderate contention levels (i.e., $0.4 \leq$ skew factor < 0.6), the abort rate of all protocols increases sharply, and still, RedT outperforms the others. The reason why MDCC deteriorates the most is that, besides the increasing abort overhead, it requires two additional round-trips for the primary replica to step in and resolve the conflict. For reference, we report the average/maximum number of inter-DC communications/round-trips for MDCC in low/high contention scenarios in Table 2. Under high contention levels (i.e., skew factor ≥ 0.6), as the growing number of aborts takes the dominant influence, all protocols suffer from severe performance degradation.

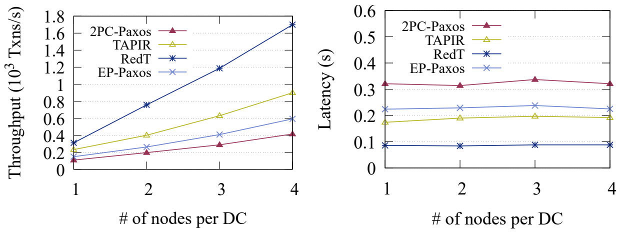
6.5 Impact of RDMA

We first compare the performance of RedT and RedT without RDMA, which replaces RDMA networks with TCP/IP networks and is similar to the idea of multi-level 2PC [37]. The results are given in Figure 12(a). For low or moderate inter-DC network delays, i.e., when the data centers involved are not so far away, RedT significantly outperforms RedT without RDMA. For high inter-DC network delays, however, their throughput converges, as the contribution of RDMA grows negligible compared to the bottlenecked inter-DC networks. As a result, RDMA is shown to be effective when the replicas are deployed in nearby data centers.

To show the efficiency of our RDMA-based implementation, we then compare RedT with a recent RDMA-based protocol DSLR [58]. Each partition is set to have one replica and all transactions are intra-DC transactions. We report the result in Figure 12(b). Similar to RedT, DSLR uses 2PL for concurrency control, and we omit the comparison with other optimistic RDMA protocols [14, 25, 54] for fairness. We observe that the performance of RedT is generally comparable to that of DSLR, at a significantly higher level than traditional transaction processing protocols. Although the performance of RedT grows slightly lower with an increasing number of threads, this cost is acceptable as RedT requires additional overhead to provide fault tolerance guarantees.

6.6 System Scalability

We study the system scalability over both YCSB and TPC-C, by varying the number of nodes in a data center from 1 to 4. We plot the results over YCSB in Figure 13. As we can see, when the number of nodes per data center increases, the throughput of all protocols increases almost linearly, while the latency remains almost



(a) Throughput (b) Latency

Figure 15: Scalability - TPC-C (Payment)

unchanged. This is because we set each transaction to read/write primary replicas across two data centers, and varying the number of nodes per data center does not affect the execution of the transaction. With a growing number of nodes, there is an obvious benefit of RedT compared with the other protocols. The reason is that RedT performs constantly the best for each node, and this accumulated benefit is amplified by increasing the number of nodes. Figure 14 and 15 report the scalability evaluation over TPC-C New-order and Payment, respectively. Due to the same reasons, all protocols enjoy a linear increase in throughput and an unchanged latency, and again, RedT is superior to the other three protocols.

7 RELATED WORK

Substantial efforts have been devoted to transforming distributed transaction processing into local transaction processing. For these works, they focus on carefully designing a static or dynamic data partitioning scheme [1, 2, 12, 16, 39, 42, 44, 47, 51, 61] so that all data partitions that each transaction accesses are on the same node. Yet, a static scheme works only if the optimal data placement is known a priori and never changes. Under the dynamic data partitioning scheme, transactions are executed in batches. For each distributed transaction, before it starts to execute, all the primary replicas that it reads/writes must be moved to the same node in advance. However, in this case, frequent data migration overhead across data centers could lead to significant performance degradation. As a remedy, an alternative method [34] is to arrange one super node that takes all replicas of the whole data partitions. All distributed transactions in a batch are then assigned to this super node, and before any of them starts to execute, all involved secondary replicas in this node are promoted as primary replicas so that they are transformed to local transactions [34]. Nevertheless, for this method, the super node may become the bottleneck when the workloads are skewed or the size of the whole data partitions is prohibitively huge.

As opposed to eliminating distributed transactions, quite a few works attempt to optimize distributed transaction processing by reducing the number of network round-trips in conventional networks [57]. Multi-level 2PC [37] reduces the expensive inter-DC communication by layering participants, but at the cost of a higher intra-DC coordination overhead. Early Prepare [46] further eliminates the barrier between the execution and prepare phase, where each participant enters the prepare phase after the read/write execution without waiting for the notification from the coordinator. Parallel Commit [52] turns some synchronous round-trips to asynchronous and is specially designed for MVCC-based write intents in CockroachDB. A series of classic algorithms are deterministic-based [17–19, 32, 33, 40, 41, 49] with an assumption that the read/write set of each transaction is known in advance. Commutativity is a different constraint from determinism, with which MDCC [28] is able to commit in a single synchronous round-trip. Without commutativity of operations and in case of concurrent updates, however, primary replicas must step in to resolve conflicts in two additional round-trips. Recent works like TAPIR [63], G-PAC [35] attempt to unify the commitment and consensus protocols in a single framework. Specifically, instead of synchronizing log replications from primary replicas to secondary replicas, in the prepare/commit phase, the coordinator directly sends log replications to secondary replicas, reducing the number of round-trips. Compared with these works, RedT concentrates on the reduced number of not only network round-trips, but also communication per round trip, and proposes a pre-write-log mechanism that is able to overlap the prepare and execution phase and eliminate redundant synchronization.

It has been a hotspot to use RDMA to optimize the distributed transaction processing. As a foundation, a set of works [8, 26, 56, 60, 65] provide insights and guidelines for RDMA usage tips like comparison of one-sided and two-sided verbs, requests sending/receiving optimizations, and system architecture designs. On account of its tempting features, RDMA is widely used to optimize distributed transaction processing. Dozens of works utilize RDMA to provide high availability with replicas [10, 15, 62], generate global timestamps with monotonicity [45], or reimplement concurrency control algorithms including OCC-based variants [14, 54] and 2PL-based variants [6, 55, 58]. As all these works confine their application scenarios into a single data center, RedT presents a first-of-its-kind attempt to support inter-data-center transaction processing and data center fault tolerance simultaneously.

Other works [3, 4, 13, 22, 24] target to reduce the cost of network communications in BFT-based (Byzantine Fault Tolerant) blockchain systems, As we target the optimization for CFT-based (Crash Fault Tolerant) database systems, the extension of these works to RedT would be considered as our future work.

8 CONCLUSIONS

In this paper, we present RedT, a novel distributed transaction processing protocol that supports inter-data-center transaction processing and data center fault tolerance simultaneously. RedT extends 2PC by decomposing transactions into sub-transactions in terms of the data center granularity, and proposing a pre-write-log mechanism that eliminates the log synchronization in the prepare phase. It is particularly designed for high-latency and unstable inter-DC networks, and achieves high performance by reducing the number of inter-DC round-trips. Through extensive experiments on YCSB and TPC-C, RedT is proved to excel over other state-of-the-art methods by providing 1.57× better throughput, 0.56× lower latency, as well as stabler performance in most cases.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (Number 61972403, 61732014) and Ant Group Research Fund.

REFERENCES

[1] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. DynaMast: Adaptive Dynamic Mastering for Replicated Systems. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020.* IEEE, 1381–1392. https://doi.org/10.1109/ICDE48307.2020.00123

[2] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. 2020. MorphoSys: Automatic Physical Design Metamorphosis for Distributed Database Systems. *Proc. VLDB Endow.* 13, 13 (oct 2020), 3573–3587. https://doi.org/10.14778/3424573.3424578

[3] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. CAPER: A Cross-Application Permissioned Blockchain. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1385–1398. https://doi.org/10.14778/3342263.3342275

[4] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 76–88. https://doi.org/10.1145/3448016.3452807

[5] Vlad Barshai, Yvonne Chan, Hua Lu, Satpal Sohal, et al. 2012. *Delivering continuity and extreme capacity with the IBM DB2 pureScale feature.* IBM Redbooks.

[6] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong Consistency is Not Hard to Get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *Proc. VLDB Endow.* 12, 13 (sep 2019), 2325–2338. https://doi.org/10.14778/3358701.3358702

[7] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (jun 1981), 185–221. https://doi.org/10.1145/356842.356846

[8] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (mar 2016), 528–539. https://doi.org/10.14778/2904483.2904485

[9] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2477–2489. https://doi.org/10.1145/3448016.3457560

[10] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. 2016. Fast and General Distributed Transactions Using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 26, 17 pages. https://doi.org/10.1145/2901318.2901349

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[12] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-Driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1-2 (sep 2010), 48–57. https://doi.org/10.14778/1920841.1920853

[13] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 123–140. https://doi.org/10.1145/3299869.3319889

[14] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}

[15] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. *No Compromises: Distributed Transactions with Consistency, Availability, and Performance.* Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425

[16] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 299–313. https://doi.org/10.1145/2723372.2723726

[17] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.* 8, 11 (July 2015), 1190–1201. https://doi.org/10.14778/2809974.2809981

[18] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (jan 2017), 613–624. https://doi.org/10.14778/3055540.3055553

[19] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. 2014. Lazy Evaluation of Transactions in Database Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2588555.2610529

[20] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. 2021. When Cloud Storage Meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 519–533. https://www.usenix.org/conference/nsdi21/presentation/gao

[21] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 202–215. https://doi.org/10.1145/2934872.2934908

[22] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (mar 2020), 868–883. https://doi.org/10.14778/3380750.3380757

[23] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (jan 2017), 553–564. https://doi.org/10.14778/3055540.3055548

[24] Jelle Hellings and Mohammad Sadoghi. 2021. ByShard: Sharding in a Byzantine Environment. *Proc. VLDB Endow.* 14, 11 (oct 2021), 2230–2243. https://doi.org/10.14778/3476249.3476275

[25] Masoud Hemmatpour, Bartolomeo Montrucchio, Maurizio Rebaudengo, and Mohammad Sadoghi. 2022. Analyzing In-Memory NoSQL Landscape. *IEEE Transactions on Knowledge and Data Engineering* 34, 4 (2022), 1628–1643. https://doi.org/10.1109/TKDE.2020.3002908

[26] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[28] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2465351.2465363

[29] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (may 1998), 133–169. https://doi.org/10.1145/279227.279229

[30] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (October 2006), 79–103. https://www.microsoft.com/en-us/research/publication/fast-paxos/

[31] Network Latency. 2022. http://ipnetwork.windstream.net/.

[32] Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. 2021. Don't Look Back, Look into the Future: Prescient Data Partitioning and Migration for Deterministic Database Systems. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1156–1168. https://doi.org/10.1145/3448016.3452827

[33] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2047–2060. https://doi.org/10.14778/3407790.3407808

[34] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1316–1329. https://doi.org/10.14778/3342263.3342270

[35] Sujaya Maiyya, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2019. Unifying Consensus and Atomic Commitment for Effective Cloud Data Management. *Proc. VLDB Endow.* 12, 5 (jan 2019), 611–623. https://doi.org/10.14778/3303753.3303765

[36] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 103–114. https://www.usenix.org/conference/atc13/technical-sessions/presentation/mitchell

[37] C. Mohan, B. Lindsay, and R. Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.* 11, 4 (dec 1986), 378–396. https://doi.org/10.1145/7239.7266

[38] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[39] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2213836.2213844

[40] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication.. In *EDBT*. 73–84.

[41] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2021. Caracal: Contention Management with Deterministic Concurrency Control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 180–194. https://doi.org/10.1145/3477132.3483591

[42] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. 2013. SWORD: Scalable Workload-Aware Data Placement for Transactional Workloads. In *Proceedings of the 16th International Conference on Extending Database Technology* (Genoa, Italy) *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 430–441. https://doi.org/10.1145/2452376.2452427

[43] RedT. 2022. https://github.com/rhaaaa123/RedT.git/.

[44] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-Grained Adaptive Partitioning for General Database Schemas. *Proc. VLDB Endow.* 10, 4 (nov 2016), 445–456. https://doi.org/10.14778/3025111.3025125

[45] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojević, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 433–448. https://doi.org/10.1145/3299869.3300069

[46] J.W. Stamos and F. Cristian. 1990. A low-cost atomic commit protocol. In *Proceedings Ninth Symposium on Reliable Distributed Systems*. 66–75. https://doi.org/10.1109/RELDIS.1990.93952

[47] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.* 8, 3 (nov 2014), 245–256. https://doi.org/10.14778/2735508.2735514

[48] Robert H. Thomas. 1979. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Trans. Database Syst.* 4, 2 (jun 1979), 180–209. https://doi.org/10.1145/320071.320076

[49] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[50] TPC-C. 2022. http://www.tpc.org/tpcc/.

[51] Khai Q. Tran, Jeffrey F. Naughton, Bruhathi Sundarmurthy, and Dimitris Tsirogiannis. 2014. JECB: A Join-Extension, Code-Based Approach to OLTP Data Partitioning. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/2588555.2610532

[52] Nathan VanBenschoten. [n.d.]. *Parallel Commits: An Atomic Commit Protocol For Globally Distributed Transactions.* https://www.cockroachlabs.com/blog/parallel-commits/ (2019, November 7).

[53] C. Wang and X. Qian. 5555. RDMA-enabled Concurrency Control Protocols for Transactions in the Cloud Era. *IEEE Transactions on Cloud Computing* PP, 01 (sep 5555), 1–1. https://doi.org/10.1109/TCC.2021.3116516

[54] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. 2018. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better!. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 233–251. https://www.usenix.org/conference/osdi18/presentation/wei

[55] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 87–104. https://doi.org/10.1145/2815400.2815419

[56] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Characterizing and Optimizing Remote Persistent Memory with RDMA and NVM. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 523–536. https://www.usenix.org/conference/atc21/presentation/wei

[57] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: A 707 Million TpmC Distributed Relational Database System. *Proc. VLDB Endow.* 15, 12 (sep 2022), 3385–3397. https://doi.org/10.14778/3554821.3554830

[58] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. 2018. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1571–1586. https://doi.org/10.1145/3183713.3196890

[59] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (nov 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[60] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.* 10, 6 (feb 2017), 685–696. https://doi.org/10.14778/3055330.3055335

[61] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. 2015. Locality-Aware Partitioning in Parallel Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/2723372.2723718

[62] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1637–1650. https://doi.org/10.14778/3342263.3342639

[63] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 263–278. https://doi.org/10.1145/2815400.2815404

[64] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 523–536. https://doi.org/10.1145/2785956.2787484

[65] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-Based Index Structures for Fast RDMA-Capable Networks. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 741–758. https://doi.org/10.1145/3299869.3300081