Achieving Low Tail-latency and High Scalability for Serializable Transactions in Edge Computing

Xusheng Chen
The University of Hong Kong
xschen@cs.hku.hk

Haoze Song
The University of Hong Kong
hzsong@cs.hku.hk

Jianyu Jiang
The University of Hong Kong
jyjiang@cs.hku.hk

Chaoyi Ruan
USTC
rcy@mail.ustc.edu.cn

Cheng Li
USTC
chengli7@ustc.edu.cn

Sen Wang
Huawei
wangsen@huawei.com

Gong Zhang
Huawei
nicholas.zhang@huawei.com

Reynold Cheng
The University of Hong Kong
ckcheng@cs.hku.hk

Heming Cui[*]
The University of Hong Kong
heming@cs.hku.hk

Abstract

A distributed database utilizing the wide-spread edge computing servers to provide low-latency data access with the serializability guarantee is highly desirable for emerging edge computing applications. In an edge database, nodes are divided into regions, and a transaction can be categorized as intra-region (IRT) or cross-region (CRT) based on whether it accesses data in different regions. In addition to serializability, we insist that a practical edge database should provide low tail latency for both IRTs and CRTs, and such low latency must be scalable to a large number of regions. Unfortunately, none of existing geo-replicated serializable databases or edge databases can meet such requirements.

In this paper, we present Dast (Decentralized Anticipate and STretch), the first edge database that can meet the stringent performance requirements with serializability. Our key idea is to order transactions by anticipating when they are ready to execute: Dast binds an IRT to the latest timestamp and binds a CRT to a future timestamp to avoid the coordination of CRTs blocking IRTs. Dast also carries a new stretchable clock abstraction to tolerate inaccurate anticipations and to handle cross-region data reads. Our evaluation shows that, compared to three relevant serializable databases, Dast's 99-percentile latency was 87.9%~93.2% lower for IRTs and 27.7%~70.4% lower for CRTs; Dast's low latency is scalable to a large number of regions.

[*]Corresponding author.

EuroSys '21, April 26–28, 2021, Online, United Kingdom
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8334-9/21/04.
https://doi.org/10.1145/3447786.3456238

CCS Concepts • **Information systems → Distributed database transactions**;

Keywords edge computing, distributed transaction, tail-latency, scalability

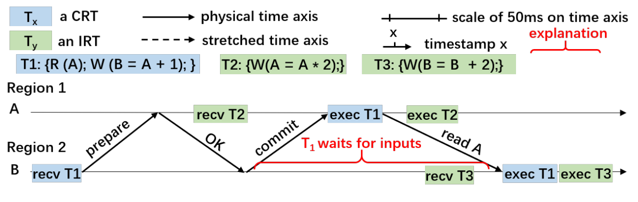
ACM Reference Format:
Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. 2021. Achieving Low Tail-latency and High Scalability for Serializable Transactions in Edge Computing. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–28, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3447786.3456238

1 Introduction

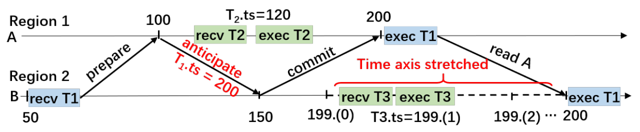
The prosperity of edge computing fosters the deployment of distributed databases [35, 59, 88] on edge nodes [3, 7, 11]. An edge node can provide fast data access for clients in a town or a city [91] as the network latency from a client to an edge node is usually less than 10ms [91], much smaller than the latency to a conventional data center (up to 100ms [92]). Hereinafter, we call a geo-distributed database running in conventional data centers (e.g., Janus [73], Tapir [113], SLOG [86], Calvin [99], Carousel [109], Spanner [27]) a *conventional database* and call a database running on edge nodes (e.g., SEQ [59] and T-Cache [35]) as an *edge database.*

So far, edge databases [35, 59] mainly serve as web-browsing caches [82] and provide weak consistency (isolation) guarantees [21]. However, the increasingly fast network [62] enables the deployment of mission-critical applications on edge nodes (e.g., AR/VR [84, 87], smart-city management [53, 71, 103], and industrial IoT systems [29, 85, 107]). Due to the rigorous correctness [15, 16, 90] and usability requirements of these applications, this paper aims to develop an edge database that provides the serializability guarantee, where transactions are equivalent to be performed in a serial order.

In this paper, we summarize three characteristics of an edge database. First, an edge database usually encounters



(a) The FCFS approach (e.g., Janus [73]) causes CRTs to block IRTs.



(b) The anticipated and stretch approach satisfies **R1** and **R2** simultaneously.

Figure 1. A simple example comparing the FCFS approach and Dast, without considering replication and clock skewness.

workloads with good spatial locality. An edge database typically groups edge nodes into *regions* [11, 75], each containing a number of edge nodes and clients connected with a fast network. Each region holds a number of database partitions (i.e., shards) frequently accessed by clients in this region. We call a transaction an *intra-region transaction* (IRT) if the client only accesses shards only in its own region; otherwise, we call it a *cross-region transaction* (CRT). Recent work [14, 18, 59, 75] shows that, in a typical edge workload, a great majority (e.g., 90% [14]) of transactions are IRTs.

Second, an edge database often serves mission-critical applications with stringent latency requirements. For an IoT-powered industrial management system [29, 85, 107], more than 99% of IRTs (e.g., intra-site robot collaboration) should finish within a few tens of milliseconds, and CRTs (e.g., remote maintenance and monitoring) should finish within a few hundreds of milliseconds. Third, an edge database can involve hundreds or thousands of regions [18, 75] to serve clients on a large scale.

These characteristics can be distilled into three crucial technical requirements for an edge database. First (**R1**), when an IRT conflicts with a CRT (e.g., has read-write dependencies), the execution of the IRT should hardly be blocked by the cross-region communication of the CRT. Second (**R2**), a CRT T_c should not be aborted due to conflicts if no node is suspected to have failed [68, 79]. Otherwise, as the cross-region communication time for T_c can be hundreds of milliseconds, much longer than the finish time of IRTs, T_c may starve due to repeated aborting if conflicting IRTs continuously arrive. Third (**R3**), the edge database's latency and throughput should be scalable to a large number of regions.

Unfortunately, to the best of our knowledge, although many systems improve the performance of conventional databases, none of them can meet these three requirements simultaneously if deployed on edge nodes. We believe a key reason is that existing conventional databases schedule transactions using a first-come-first-serve (FCFS) approach based on their arriving orders at replicas [27, 36, 46, 73, 109, 113] or a central server [86, 99].

Figure 1a explains two main scenarios of IRTs being blocked by CRTs in an FCFS system. First, node A receives an IRT T_2 after preparing a CRT T_1, so T_2 is ordered after T_1 and is blocked. Second, the execution of T_1 on node B needs to wait for a cross-region data read, so the execution of the subsequent IRT T_3 is blocked. As the cross-region latency can be tens of times longer than the intra-region latency, such blockings can easily cause a long tail latency for IRTs.

This paper presents Dast (Decentralized Anticipate and STretch), a new approach that can enforce serializability and all three requirements. Dast uses timestamps to order transactions, and our key idea to remedy the deficiency of the FCFS approach is assigning timestamps to transactions based on when they are *ready to execute*. Specifically, when a node receives a transaction from a client, if the transaction is an IRT, the node (i.e., the transaction's coordinator) assigns its *latest* timestamp to the IRT for low latency. If the transaction is a CRT, the coordinator assigns the CRT a *future* timestamp to avoid its coordination blocking subsequent IRTs. As shown in Figure 1b, the CRT T_1 is assigned a future timestamp 200 while the late-arriving IRT T_2 is assigned timestamp 120, ordered before T_1 without being blocked (**R1**).

Dast carries a new two-phase decentralized anticipation protocol (2DA) that integrates the selection of a feasible future timestamp for a CRT to the two-phase commit (2PC) protocol [83] for committing the CRT. Dast's 2DA protocol can commit a CRT without aborts even if nodes' clocks are not synchronized, provided that no node is suspected to have failed (**R2**). Meanwhile, the 2DA protocol achieves the minimal number of round trips for committing a CRT.

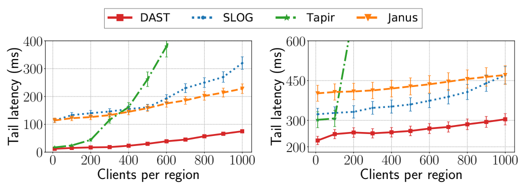
Even with the 2DA protocol, it is still challenging to ensure **R1**. Consider the CRT T_1 with cross-region data reads in Figure 1. T_1's execution on node B must be half cross-region round trip time after T_1's execution on node A; however, for serializability, these two executions should have the same timestamp, inevitably blocking subsequent IRTs on node B.

Dast addresses this challenge with a new hybrid clock with stretchable timestamp granularity. As shown in Figure 1b, when node B's clock approaches 200, but T_1 is still waiting for cross-region inputs, node B stretches the granularity of its local clock to make its clock grow slowly until T_1's input is ready. By doing so, T_3 can be assigned a timestamp 199.(1) ordered before T_1, so it is not blocked. Meanwhile, the stretchability of Dast's hybrid clock also handles the inaccuracy of anticipation on the asynchronous Internet.

We implemented Dast on the Janus [73] codebase, a mature modular framework for evaluating distributed databases. We compared Dast to three relevant serializable databases, Janus [73], Tapir [113], and SLOG [86], and we used all workloads with locality features evaluated in relevant systems [27, 36, 46, 73, 76, 86, 99, 109, 113]. Our evaluation shows that:

	DAST	Tapir [113]	Carousel [109]	Calvin [99]	Spanner [27]	Janus [73]	SLOG [86]	Ocean Vista [36]	DPaxos [75], WPaxos [18]
Serializability	✓	✓	✓	✓	✓	✓	✓	✓	✗
R1	✓	✓	✗	✗	✗	✗	✗	✗	n/a.
R2	✓	✗	✗	✓	✓	✓	✓	✓	n/a.
R3	✓	✓	✓	✗	✗	✓	✗	✗	✓

Table 1. Comparison of DAST and existing systems. "n/a." means not applicable as they do not support multi-shard transactions.



(a) 99% tail latency of IRTs. **(b)** 99% tail latency of CRTs.

Figure 2. DAST's tail latency compared to relevant databases on the standard TPC-C [14] workload (detailed setup is in §6).

- **R1:** Figure 2a shows that DAST's 99-percentile latency of IRTs is 87.9%~93.2% lower than the three systems.
- **R2:** Figure 2b shows that DAST's 99-percentile latency of CRTs is 27.7%~70.4% lower than the three systems.
- **R3:** DAST is scalable to a large number of regions. When we increased the number of regions from 10 to 100, DAST's throughput increased almost linearly, and DAST's latency for IRTs and CRTs remained stable.
- DAST's median latency and throughput were comparable to relevant systems and were robust to network anomalies.

Our major contribution is DAST, a new transaction scheduling method that enforces all the three requirements with serializability for an edge database. Our other contribution includes the DAST implementation, released on github.com/hku-systems/dast. Compared to conventional serializable databases, DAST takes the first step to meet all the three requirements, making DAST unique to support mission-critical edge computing applications. Although existing edge databases can meet these three requirements, none of them has the same serializability guarantee as DAST. DAST can facilitate porting various applications to the edge, including smart industrial plants [25, 97] and intelligent transportation systems [9, 10], discussed in §6.4.

In the rest of the paper: §2 discusses related work; §3 gives an overview; §4 shows DAST's design; §5 covers DAST's implementation; §6 shows our evaluation; §7 concludes.

2 Background and Related Work

DAST's design is motivated by the emerging mission-critical distributed edge computing applications that desire serializability and low tail-latency. For instance, a smart city management system [26, 53, 60, 110] coordinates vehicles and city facilities to conduct collaborative road lane allocation, non-stop toll payment, charging station scheduling, and real-time traffic lights management for traffic scheduling. A smart grid system [20, 58] managing the energy transmission between many power plants and users needs to collect real-time electricity consumption information from users and to adjust the power supply routes in order to provide stable voltage and fast recovery from link failures. Another example is the smart logistic system [57] which manages robotics to do goods sorting, shelf distribution, and cargo packing in each warehouse, and transports goods among many warehouses to maintain sufficient stock level. DAST's low tail latency and serializability are desirable properties for these applications.

Edge databases. Much work [18, 35, 59, 75, 88] leverages edge nodes to provide fast data access to clients. However, unlike DAST, they either do not support multi-shard transactions [18, 75] or provide a weaker isolation level than serializability [35, 59, 88]. Therefore, our evaluation did not compare DAST with these systems (§6).

DPaxos [75] and WPaxos [18] are two Paxos-based [49, 51] protocols that achieve low-latency data access and fast leader elections for a key-value store. DAST is complementary to them as DAST focuses on serializable transactions.

SEQ [59] provides snapshot isolation [21] to a data cache on edge nodes; T-Cache [35] defines a new weak isolation level tailored for web caches; EdgeX [88] determines which components of a web service should be deployed on edge nodes. Gesto [17] proposes a novel hierarchical architecture for causally consistent partially replicated edge data stores that support fast client migrations. These systems are orthogonal to DAST because they ensure only weak isolation, while DAST ensures transaction serializability.

Conventional geo-distributed databases. There are many influential serializable conventional databases [27, 36, 46, 73, 76, 86, 99, 109, 113], but they cannot meet the three crucial technical requirements (Table 1) if deployed with the partial replication [55, 56, 109] manner (i.e., each region does not hold all database shards) on edge computing nodes.

Existing conventional databases lie in two categories. The first category [46, 77, 109, 113] takes the deferred update (DU) [42] approach, where a transaction is executed locally at a coordinator and then certified globally. DU databases can achieve high throughput on workloads with low conflict rates. However, DU databases cannot meet **R2** because a CRT is aborted if it conflicts with any other transactions.

Tapir [113] and Carousel [109] are two latest DU databases. Tapir can meet **R1** because an IRT is never blocked by CRTs. However, the IRT itself may be retried multiple times. Carousel lets a replica reject an incoming transaction if it conflicts with a prepared one to avoid deadlocks, so Carousel cannot meet **R1**. We selected Tapir as our baseline representing DU databases because Tapir meets one more requirement than Carousel, and Carousel is not open source.

The second category (e.g., Calvin [99], SLOG [86], Spanner [27], and Janus [73]) takes the state machine replication (SMR) approach, where nodes first agree on the order of all transactions and then deterministically execute them. SMR-based databases meet **R2** because they do not abort a CRT due to conflicts, but they cannot meet **R1** because they order all transactions using the FCFS approach. An IRT may be blocked by (1) the coordination processes of CRTs arriving before it, for which we call "coordination blocking" (e.g., T_2 in Figure 1), and (2) a CRT waiting for a cross-region read, for which we call "dependency blocking" (e.g., T_3 in Figure 1). DAST also takes the SMR approach, but DAST avoids coordination blockings by anticipation and avoids the dependency blockings by its new stretchable clock. Below we briefly introduce latest SMR-based database systems.

Janus [73] has a fast path and a slow path. Each node holds a dependency graph of incoming transactions. A transaction T's coordinator collects T's dependency graphs from participating nodes and commits T using the fast path if the consolidated graph is acyclic; otherwise (slow path), it cuts the graph into acyclic subgraphs and sends them to participating nodes. Janus has both coordination blockings and dependency blockings (violating **R1**), but Janus meets **R3** because committing a CRT involves only participating regions.

SLOG [86] leverages data locality to achieve low mean latency for IRTs. SLOG uses a global ordering service [49] to order all CRTs, and each region has a leader that orders all transactions accessing this region. SLOG cannot meet **R1** due to execution blockings (admitted in its paper). Moreover, SLOG's global ordering service is a scalability bottleneck (**R3**), as it not only orders all CRTs regardless of whether they conflict, but also needs to send each ordered log entry to all regions. Otherwise, a region missing an entry cannot determine whether the entry is irrelevant or lost.

Ocean Vista (OV) assigns each transaction a timestamp and maintains a global watermark: transactions with timestamps below the watermark can be executed. OV has coordination blockings (**R1**): an IRT T_i waits for at least one cross-region RTT before the watermark passes T_i's timestamp. OV cannot meet **R3**: for a hundred regions, OV needs hundreds of nodes with $O(n^2)$ message complexity for maintaining the watermark. Therefore, OV is not suitable for edge databases, and our evaluation did not include OV.

Hybrid clocks in databases. DAST is not the first database using hybrid clocks. TicToc [111] presents a late-binding approach using stretchable timestamps to reduce abort rates in an optimistic [48] database. Unlike DAST, TicToc is not geo-distributed. Clock-SI [33] proposes to assign timestamps that are smaller than the nodes' physical clocks by a changeable value to increase transaction commit rates. Unlike DAST, Clock-SI provides only snapshot isolation. Eunomia [38] uses hybrid clocks to build a geo-distributed causally consistent key-value store, while DAST is a serializable edge database.

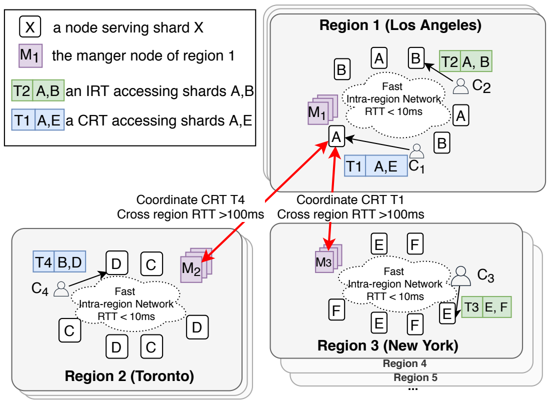

X a node serving shard X
M1 the manger node of region 1
T2 A,B an IRT accessing shards A,B
T1 A,E a CRT accessing shards A,E
Region 1 (Los Angeles)
T2 A, B
C2
C1
T1 A,E
M1
Fast Intra-region Network RTT < 10ms
Coordinate CRT T4 Cross region RTT >100ms
Coordinate CRT T1 Cross region RTT >100ms
T4 B,D
C4
M2
Region 2 (Toronto)
M3
C3
T3 E, F
Region 3 (New York)
Region 4
Region 5
...

Figure 3. DAST's architecture on edge computing nodes.

DAST's timestamps have the same format as HLC [47], but their purposes and the ways for maintaining clocks are different. HLC is used for tracking causal dependency [50] among events, and its frac part increases upon events' arrival. DAST's timestamps' frac parts are used to assign IRTs timestamps smaller than prepared CRTs' to avoid IRTs getting blocked.

Anticipation in consensus protocols. Existing work uses the anticipation merit to improve the performance of consensus protocols. Sousa et al. [94] propose a new delay compensation approach that can optimistically commit a request if the network ordering is preserved. Domino [108] leverages the stable latency of dedicated links among data centers to assign requests to different log entries. Unlike DAST, these systems have no transactional support.

Databases in a single data center. Existing work uses transaction reordering [31, 34, 64] to reduce abort rates in optimistic databases. Many systems, including STAR [63], MOCC [104], ERMIA [44], Maat [65], E-Store [96], Polaris [45], and H-Store [43], improve the performance of databases running in a data center. All these systems are not geo-distributed and are orthogonal to DAST.

3 Overview

3.1 System Model

Figure 3 shows DAST's deployment model. Each node is an edge server holding a database shard. A region contains multiple nodes and clients connected with fast networks: the intra-region round trip time (RTT) is much lower (e.g., less than ten milliseconds) compared to the cross-region RTT (e.g., hundreds of milliseconds) [3, 7, 11]. Nodes are equipped with loosely synchronized clocks (e.g., by NTP [70]). DAST does not rely on these clocks for consistency.

Each region has a manager to handle node failovers (§4.4) and to anticipate timestamps for CRTs accessing this region (§4.3). The manager's states are replicated by an SMR service [79] but only states off the critical path of transaction processing (e.g., node membership) are replicated.

Partial replication. DAST assigns the region that accesses a shard most frequently as the shard's host region. For the

low latency of IRTs, a shard is synchronously replicated for $2f + 1$ copies only within its host region. Nodes in a region are usually not geographically close for disaster-tolerance.
Consistency and failure model. DAST is designed to run on an asynchronous network: network packets can be dropped, reordered, or delayed arbitrarily; the network can be partitioned; nodes and clients can fail by crashing at any time. We do not consider Byzantine failures [24, 69, 102]. DAST has the following guarantees:

- **One-copy serializability.** All transactions performed in DAST are equivalent to be performed in a serial order [22], and this serial order is identical among replicas.
- **No stale reads.** Once a transaction T is executed, all transactions starting afterward see updates made by T.
- **Availability.** DAST preserves availability if no more than f replicas of any shard fail. Cross-region network partitions may affect CRTs' availability but will not affect IRTs', which is a desirable feature for an edge database. DAST preserves availability on arbitrary client failures.

3.2 DAST Protocol Overview

Notations. We use T for a transaction, n for a node, r for a region, m for a manager. We use n^r for a node in region r and T^r for a transaction submitted by a client in region r. For a CRT T_c^r, we call r its *home region* and other regions as *remote regions.* We call all transactions accessing a node n as n's *relevant transactions.* When a client submits a transaction T to a node, the node works as the *coordinator* of T.
Two-phase decentralized anticipation (2DA). DAST assigns a CRT (T_c) a future timestamp (ts) to avoids its communication blocking subsequent IRTs (**R1**). A strawman approach is letting T_c's coordinator at region r_1 determine ts by itself and send it to all T_c's participating regions. However, as nodes' clocks are not synchronized, and network packets can be delayed, when T_c arrives at a node n^{r_2} in region r_2, n^{r_2} may have already executed transactions with timestamps larger than ts, causing T_c to be aborted (violating **R2**).

DAST's 2DA protocol integrates the selection of ts to the 2PC protocol [83] for committing the CRT T_c. In the first phase, T_c's participating nodes in each region r determine a future timestamp ts^r that they anticipate to receive the `commit` message from T_c's coordinator, based on the estimated RTT; these nodes also promise not to assign timestamps larger than ts^r for any transactions until T_c is executed. In the second phase, the coordinator commits T_c with the *largest* anticipated timestamp among all participating regions. As participating nodes promise not to assign timestamps larger than ts, T_c will not be aborted if no node fails.
DAST's hybrid clock. DAST's clock (`dclock`) with stretchable granularity has two main functionalities. First, it avoids IRTs from being blocked by an inaccurately anticipated CRT due to network anomalies or clock skewness. Consider a node n^r anticipating to commit a CRT T_c at time ts^r but cannot receive the `commit` message in time; then, an IRT conflicting with T_c arrives. To ensure serializability, this node faces a dilemma: it must abort T_c (violating **R2**) or let the IRT wait until the T_c is committed (violating **R1**). Second, DAST's hybrid clock relaxes the tension between DAST's timestamps and physical time to avoid a committed CRT blocking subsequent IRTs when the CRT is waiting for input from remote regions (i.e., the dependency blocking, see §2).

In DAST, each node's `dclock` has three fields ⟨`time`, `nid`, `frac`⟩: `time` is the physical part based on the node's system clock to facilitate anticipation; `nid` is unique for each node to ensure timestamp uniqueness; `frac` is the logical part to avoid IRTs being blocked by CRTs. Initially, a node increases `time` with its system clock. When it detects that further increasing `time` will cause an IRT blocked by CRTs, it freezes `time` and increases `frac` instead, so that the timestamp assigned to the IRT is smaller than the CRTs'. Once the CRT is executed, the node resumes `time` to catch up with the node's systems clock to facilitate future anticipations.
Per-region clock tracking (PCT). DAST's PCT protocol (§4.3) is inspired by Mencius [66] and ClockRSM [32] but is different in two aspects. First, they only provide linearizability for a single object while DAST ensures serializability for multi-object transactions. Second, in these systems, a node needs to track the clocks of all other nodes, while in DAST, a node only tracks the `dclocks` of intra-region nodes.

4 DAST Runtime Protocol

4.1 DAST's Transaction Model

DAST's transaction model extends the widely-used independent transaction model [28, 41, 43, 73] with the support for *acyclic value dependencies* and *user-level conditional aborts.*

In DAST, transactions are written as stored procedures, which are also widely used in existing distributed databases [37, 43, 72, 73, 95, 101]. Each transaction consists of a set of stored procedures, referred to as *pieces.* Each piece only accesses one data shard known a prior. The execution of each piece is deterministic (i.e., only depending on inputs and current database state), and pieces accessing the same shard are executed atomically in their timestamp order.

We choose to support stored procedures instead of interactive transactions because interactive transactions' latency requirements are usually less stringent [28, 43] due to user stalls and can be served by conventional databases.
Supporting acyclic value dependencies. In DAST, a transaction T can have value dependencies: a piece p_1 accessing shard s_1 can take database values stored in another shard s_2 or output of another piece accessing s_2 as p_1's input. For instance, T can transfer all remaining balance from account y in shard s_2 to account x in shard s_1, with p_1's logic being $x = x + y$. If s_1 and s_2 are in different regions, we say T contains a value dependency from the region of s_2 to the region of s_1, and DAST's transaction model requires that a

CRT does not have circular value dependencies among its accessed regions. By contrast, in the original independent transaction model, no value dependency is allowed.

Adding support for value dependency is essential for edge databases. Conventional databases can preclude such dependencies by vertical replication [74], which replicates dependent columns (of all shards) to all nodes. However, vertical replication is unsuitable for edge databases because it forsakes data locality and requires all nodes to update the replicated column synchronously on every change.

DAST uses a push-based mechanism to handle value dependencies. In the previous example, instead of letting p_1 read y during its execution phase, DAST lets a piece (e.g., p_2) on s_2 read the required value y and send it to n_1. This design is crucial for DAST to achieve low tail latency for IRTs (**R1**). Otherwise, if s_1 and s_2 are in different regions, when p_1 does the cross-region read, it blocks subsequent IRTs (violating **R1**). In DAST, p_1 does not start to execute until it receives all remote read values, and DAST's hybrid clock orders subsequent IRTs before T during such waiting, achieving both **R1** and **R2**. We say a transaction's "input is ready" on node n, when n receives all remote-shard values and can start to execute. As the dependency graph is acyclic, this mechanism will not cause a circular waiting.

Supporting conditional aborts. DAST supports conditional aborts where a transaction T determines whether to apply its writes based on read values (e.g., aborting a money transfer due to out-of-balance). For atomicity, all participating nodes must consistently agree on whether to apply T's writes.

To avoid the necessity of another round of voting [106] on whether to apply T's writes, DAST lets the developer rewrite T's stored procedures by adding explicit reads to the dependent values at all T's participating nodes. By doing so, when T is executed, all participating nodes will read identical dependent values (due to serializability) and consistently determine whether to perform data writes. Note that in DAST's transaction model, T's value dependencies must still meet the acyclic requirements after the explicit data reads are added.

All workloads evaluated by us (i.e., TPC-C [14] and TPC-A [13]) and relevant systems (e.g., YCSB [30], Retwis [52]), and typical mission-critical edge applications [53, 61, 71, 78, 87, 103] belong to DAST's transaction model. For transactions whose accessed shards are determined by read values, DAST can support them using a known technique [43, 73] that divides each transaction into a read-only transaction and a conditional-write transaction. In the presence of such transactions, DAST can meet **R1** and **R3** for these transactions and all three requirements for other transactions.

4.2 Intra-region Transactions

Each node maintains two queues in the timestamp order for unexecuted relevant transactions. The `readyQ` contains two types of transactions; (1) all received IRTs, and (2) all committed CRTs. The node will check whether each transaction

Algorithm 1: Coordinate an IRT

```
vid ← current view of this node
myTxns[] ← txns coordinated by this node in ts order
notifiedTs[n] ← the max timestamp of transactions
  coordinated by this node delivered to node n
function CreateTs():
    ret ← ⟨now() + offset, nid, 0 ⟩                    // the dclock
    if ! waitQ.empty() && waitQ.head().ts < ret :
        ret ← lastTs          // ensure the IRT with ret is not blocked
        ret.frac++  // fallback to lastTs; increase frac for monotonicity
    return lastTs ← ret
function CoordIRT(txn):
    ts ← CreateTs()
    readyQ.insert(⟨ts, txn⟩)
    myTxns.insert(⟨ts, txn⟩)
    for n in txn.partcipatingNodes do
        send(n, ⟨prepare, txn, ts, vid, GetNotifyTxns(n, ts)⟩)
          // GetNotifyTxns(n, ts) returns txns in myTxns[] with
          timestamps in the range of ( notifiedTs[n], ts ]
    when received ack from node n:
        notifiedTs[n] ← max(ts, notifiedTs[n])         // on delivery
        if majority ack from each participating shard of txn :
            txn.status ← committed
            send ⟨commit, txn, vid ⟩ to each participating node
function CheckAndExecuteTxns():   // DAST's PCT protocol
    for txn in readyQ in timestamp order do
        if txn.status != committed : break
        if maxTs[n] > txn.ts for each n :
            Execute(txn)
            readyQ.erase(txn)
        else: break
```

can be executed using DAST's PCT protocol. Note that once a CRT is committed, it is put into the same priority queue as IRTs, so DAST will not cause CRTs to starve.

The `waitQ` contains only CRTs, and DAST uses this queue to guide the stretching of its `dclock`: if the node's `dclock` passes the timestamps of any CRT in the `waitQ`, subsequent IRTs will be blocked. This queue contains two types of CRTs: (1) prepared CRTs associated with their anticipated timestamps, and (2) committed CRTs whose inputs are not ready, associated with their commit timestamps.

Algorithm 1 shows DAST's protocol for IRTs. On receiving an IRT T_i, the coordinator assigns it a timestamp ts_i using the `CreateTS` function (Line 4), which ensures that ts_i is smaller than the timestamps of all CRTs in the `waitQ`, so that T_i is not blocked by CRTs (**R1**). Then, the coordinator asks for ACKs from majority replicas of each participating shard and sends a `commit` message to all participating nodes.

In DAST, the key invariant for correctness is that when a node n executes a committed transaction T_i, it must have executed all relevant transactions with timestamps smaller than ts_i (§4.5). DAST's PCT protocol efficiently ensures this invariant. To ease understanding, we assume that all transactions are IRTs in this subsection. In DAST, when any intra-region node n_x sends a protocol message to n, it also notifies n two information: (1) its current `dclock` value, and (2) the IDs of all transactions that n_x coordinates and accesses n.

By doing so, when n is aware that the `dclock` of n_x has passed ts_i, n knows the IDs of all relevant transactions with timestamps smaller than ts_i coordinated by n_x. Therefore, n maintains an array `maxTs` for each intra-region node's max `dclock` value that n is aware of; n can execute T_i if all entries of the array are larger than ts_i. To reduce repeated notifications, DAST adopts an optimization similar to TCP's acknowledgment mechanism: each node n maintains the max *delivered* (Line 17) notification timestamp to each other node.

Calibrating dclocks among intra-region nodes. To minimize IRT delays caused by clock skewness, the `time` field of a node's `dclock` ahead of the node's system clock by an adjustable `offset`. On receiving an intra-region message with piggybacked notification timestamp ts, a node increases its `offset` to lets its `dclock` pass ts, so an IRT does not need to wait additional time for this node's `dclock` passing ts. This optimization lets the `dclocks` of a region's nodes keep pace with the fastest one, but it maintains monotonicity.

4.3 Cross-region Transactions

DAST lets each region's manager anticipate timestamps for all CRTs accessing this region, which is essential for **R2** because it gives a preliminary order to all CRTs accessing this region. Otherwise, if CRTs interleaves differently at their participating nodes, their coordinators have to abort some of them [113] or use additional round trips to reorder them [73].

Algorithm 2 shows DAST's 2DA protocol for committing a CRT T_c. In the first phase (Line 1~5), the coordinator sends a `prep-remote` message to the manager of each participating region; the manager m^r of a region r anticipates a future timestamp ts_c^r for when the region can receive the `commit-remote` message for T_c. This anticipation is based on the average RTT of recent communication between region r and the region of the coordinator. Then m^r dispatches T_c to T_c's participating nodes in its region.

On receiving T_c, a participating node n^r puts T_c into its `waitQ` to prevent its `dclock` passing ts_c^r and aborting T_c (**R2**). Meanwhile, DAST's stretchable `dclock` ensures subsequent IRTs are not blocked by T_c (**R1**).

A subtlety is that node n^r also notifies all intra-region nodes of the existence of T_c (Line 18). Otherwise, if a node not accessed by T_c assigns an IRT T_i with a timestamp $ts_i > ts_c^r$, T_i may be blocked as executing it requires the `dclocks` of all nodes to pass ts_i, but T_c's participating nodes' `dclocks` are paused at ts_c^r. This notification effectively avoids such blockings because ts_c^r is usually ahead of these nodes' `dclocks` by a cross-region RTT, so the notification can usually be delivered to all intra-region nodes before their `dclocks` passing ts_c^r, and they can stretch their `dclock` accordingly.

In the second phase (Line 6~9), after the coordinator obtains quorum ACKs from each participating shard, it commits

Algorithm 2: Coordinate a CRT

```
1  function CoordCRT(txn):
2      replicate to majority replicas of part. shards in my region
3      srcTs ← CreateTs()
4      for r in txn.participatingRegions do
5          send($m^r$, ⟨prep-remote, txn, srcTs, vid⟩)
6      when recv acks from majority replicas of each part. shard:
7          commitTs ← max(∀r, anticipateTs$^r$, CreateTs())
8          replicate to majority replicas of part. shards in my region
9          send ⟨commit-remote, txn, commitTs⟩ to part. nodes
10 function GetAnticipateTs(srcTs, srcRegion):
11     updateEstimatedRtt(srcTs, CreateTs(), srcRegion)
12     return CreateTs() + estimatedRtt[srcRegion]
13 upon receiving ⟨prep-remote, txn, srcTs, vid⟩:
14     anticipateTs$^r$ ← GetAnticipateTs(srcTs, srcRegion)
15     send ⟨prep-crt, txn, anticipateTs$^r$, vid⟩ to each part.
        node in the region of the manager
16 upon receiving ⟨prep-crt, txn, anticipateTs$^r$, vid⟩:
17     waitQ.insert(⟨anticipateTs$^r$, txn⟩)
18     myTxns.insert(CreateTs(), txn, anticipateTs$^r$)
19     send(txn.coordinator, ACK)
20 upon receiving ⟨commit-remote, txn, commitTs, vid⟩:
21     waitQ.erase(txn)
22     readyQ.insert(⟨commitTs, txn⟩)
23     if !txn.inputReady():
24         waitQ.insert(⟨commitTs, txn⟩)          // avoid blocking
25     myTxns.insert(⟨commitTs, txn⟩)          // notify other nodes
```

T_c with the *maximum* anticipated timestamp ts_c. The coordinator also quickly replicates the CRT within its region before sending out the `prep-remote` and the `commit-remote` message so that if it crashes, the manager can retrieve the coordination progress of the T_c (§4.4).

On receiving committed T_c, node n^r does two operations atomically (Line 20). First, n^r puts T_c into the `readyQ`. Second, n^r checks whether T_c's inputs are ready (§5). If so, n^r removes T_c from `waitQ`; otherwise, n^r changes T_c's timestamp from ts_c^r to ts_c in the `waitQ`, ensuring that subsequent IRTs are not blocked by T_c when T_c is waiting for inputs.

DAST's complete PCT protocol extends the IRT-only version in the previous subsection by two modifications. First, the manager occupies an entry at the `maxTs` array at each intra-region node: a committed transaction can be executed only after the `dclock` of the manager and all intra-region nodes pass the transaction's *commit* timestamp. Second, when node n receives a *committed* CRT accessing itself, n notifies all intra-region nodes of this CRT. We do not let the manager do the notification because the manager's CRT list is not replicated (§4.4) and may be incomplete after failovers.

Cross-region clock skewness. Although clock skewness will not affect DAST's correctness (§4.5), it may cause a long latency to a CRT if the CRT is assigned a large anticipated timestamp. To mitigate this problem, we extend the intra-region clock calibration mechanism (§4.2) to cross-region:

Algorithm 3: Removing suspected failed nodes.

```
1  mNodes ← member nodes in this region
2  toRemove ← [ ]                                  // suspected failed nodes
3  pCRTs, pIRTs ← uncommitted txns by nodes in toRemove
4  upon manager $m^r$ suspecting node n having failed:
5      toRemove.insert($n$)
6      mNodes.erase($n$)
7      RemoveNodes(toRemove)
8  function RemoveNodes(toRemove):
9      send ⟨remove-prep, + + vid, toRemove ⟩ to mNodes
10     when recv ⟨remove-ack, pendCRTs, pendIRTs ⟩:
11         pCRTs.insert(pendCRTs)
12         pIRTs.insert(pendIRTs)
13     if recv remove-acks from all nodes in mNodes :
14         replicate vid, pCRTs, pIRTs to backup managers
15         send ⟨remove-commit, vid, pIRTs, mNodes ⟩ to mNodes
16         for txn in pCRTs do
17             send(⟨abort, txn, vid ⟩) to partcipatingNodes
18     if timeout for remove-ack from node $n'$ in mNodes :
19         suspect $n'$ of having failed               // goto line 3
20 upon receiving ⟨remove-prep, vid, toRemove ⟩:
21     pendIRTs, pendCRTs ← prepared but uncommitted IRTs
         and CRTs coordinated by nodes in toRemove
22     reply(⟨remove-ack, pendCRTs, pendIRTs ⟩)
23 upon receiving ⟨remove-commmit, vid, pIRTs, mNodes ⟩:
24     SwitchView(vid, mNodes)
25     for relevant txn in pIRTs do
26         txn.status ← committed
```

when a node n^r receives a message from a remote region tagged with timestamp ts, it advances its `dclock` to pass $ts + \frac{1}{2}RTT$ if it lags behind, and the intra-region calibration mechanism will advance the `dclocks` for other nodes in r. We will show in §6.3 that this optimization effectively reduces CRT latency with the presence of both clock skewness and asymmetric delay among regions.

4.4 Fault Tolerance

Dast uses a *view* to represent a region's state, including nodes' membership and the leader election result of the manager. We first show how Dast handles manager failovers, then show how a manager handles normal node failovers.

Failures of managers. Dast uses an SMR service [79] to replicate each manager's states that are not on the critical path of transaction processing, including the current view and the 2PC progress for installing new views. A manager's `dclock` and handled CRTs are *not* replicated.

The SMR service is responsible for failure detections and will elect a new manager if the current manager is suspected to have failed. Then, the new manager invokes a 2PC process for (1) installing a new view with all intra-region nodes acknowledging the manager change, and (2) adjusting its `dclock` to ensure the monotonicity of anticipated timestamps assigned to CRTs. Each node carries its `maxTs` entry

Algorithm 4: Adding node n as a replica of shard s.

```
1  function AddReplica(n, s):                 // called at manager $m^r$
2      send ⟨transfer-ckpt, n⟩ to an replica of shard s
3      upon receiving ⟨install-ok, $ts_{ckpt}$⟩:
4          $ts_{ins}$ ← anticipated timestamp for finishing installing n
5          replicate $ts_{ckpt}$, $ts_{ins}$ to backup managers
6          mNodes.insert(n)
7          send ⟨add-prep, + + vid, n, $ts_{ins}$⟩ to mNodes
8          when recv ⟨add-ack⟩ from all mNodes:
9              send ⟨add-commit, vid, mNodes ⟩ to mNodes
10 upon receiving ⟨transfer-ckpt, n⟩:
11     (state, $ts_{ckpt}$) ←GenerateCheckpoint()
12     when recv delivery ack of ⟨install, state, $ts_{ckpt}$⟩ to n:
13         send ⟨install-ok, $ts_{ckpt}$⟩ to $m^r$
14 upon receiving ⟨add-prep, vid, n, $ts_{ins}$⟩:
15     waitQ.insert(⟨$ts_{ins}$, add(n)⟩) // avoid passing $ts_{ins}$
16     reply(⟨add-ack⟩)
17 upon ⟨add-commit, vid, mNodes ⟩:
18     waitQ.erase(add(n)) // remove the fake txn
19     SwitchView(vid, mNodes)
20     notifiedTs[n] ← $ts_{ckpt}$
21     maxTs[n] ← $ts_{ins}$
```

for previous managers in response to the 2PC-`prepare` message, and the new manager adjusts its `dclock` to the largest value before serving new requests.

Failures of normal nodes. To reduce the disturbance on transaction processing, Dast separates the process of *fast failover* that quickly removes a suspected failed node and *failure recovery* that asynchronously adds back replicas.

(1) Fast failover. Algorithm 3 shows the algorithm of removing suspected failed nodes. Suppose a node n is suspected to have failed (e.g., due to RPC timeouts) and is reported to the manager. Dast takes a simple policy for uncommitted transactions coordinated by n: committing all IRTs received by at least one node for low IRT latency, and aborting all CRTs because retrieving their status requires multiple rounds of cross-region communications.

When installing a new view that removes n using 2PC, the manager collects uncommitted transactions coordinated by n from all remaining nodes and these nodes discard subsequent messages from n. On receiving all replies, the manager replicates its 2PC progress to backups and sends the 2PC-commit message, which also commits all uncommitted IRTs and aborts all uncommitted CRTs coordinated by n

(2) Asynchronous failure recovery. Algorithm 3 shows the algorithm for adding a replica n as a replica of shard s. First, m lets a replica of s send a checkpoint of its current database state and the executed timestamp ts_{ckpt} to n. Then, n needs to adjust its `dclock` to be larger than the executed timestamps of all intra-region nodes, so that it will not generate transactions preceding an executed one. However, when n is adjusting its `dclock`, nodes' executed timestamps are continuously increasing.

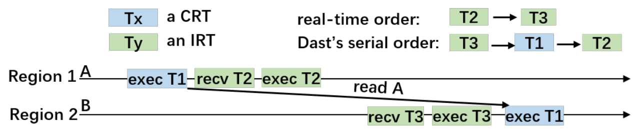


Figure 4. Dast does not preserve the real-time order of transactions accessing non-intersecting regions.

Dast addresses this challenge using a special (fake) CRT accessing all nodes in this region. The manager anticipates a future timestamp ts_{ins} that the new view should be installed. ts_{ins} can be conservatively large as admitting n is asynchronous to normal transaction processing. All nodes put this special CRT to their `waitQ`, so their `dclock`s will not pass ts_{ins} until n is admitted, and n can directly set its `dclock` to ts_{ins}.

4.5 Proof of Correctness

Dast ensures one-copy serializability (1CS) and prevents stale reads, the most severe [15, 16] anomaly for a 1CS database. Dast's guarantee is usually called strong partition serializability [16, 67].

We do not choose to provide strict serializability because its precautions are excessive for an edge database. The only possible (proved below) anomaly in Dast is breaking the out-of-band dependency of two transactions performed in non-intersecting regions (e.g., if T_2 causes T_3 via an out-of-band channel in Figure 4). Such anomalies may be fatal in conventional databases (e.g., reordering a forum post and its comment) but are rare [59] in an edge database, because these two transactions are performed by different clients and access data owned by different groups of users. Such anomalies can be avoided by letting developers write the out-of-band dependency explicitly in the transaction by adding a cross-region data read [67].

Lemma 1. *When a node n executes a transaction T with a commit timestamp ts, n has executed all relevant transactions, either IRTs or CRTs, with timestamps smaller than ts.*

Proof sketch. We first prove without node failovers. Node n's relevant transactions preceding ts include IRTs or CRTs. For IRTs, Dast's PCT protocol ensures that n is notified with all these IRTs before executing T (§4.2), and n executes transactions in timestamp order.

Consider any CRT T_1 accessing n with anticipated timestamp ts_1^r and commit timestamp ts_1 where $ts_1^r \leq ts_1 < ts$. It suffices to prove that when executing T, n is aware of T_1 with ts_1. Since T_1 is committed, quorum replicas of n's shard must have received the `prep-crt` message. If n is in the quorum, n must have received T_1's `commit` message, because otherwise its `dclock` paused before ts_1^r and cannot execute T. If n is not in the quorum, consider a replica in the quorum, that replica's `dclock` must have passed $ts > ts_1^r$ because n executing T requires all intra-region nodes' `dclock` to pass ts, and thus this replica must have committed T_1 and notified n (§4.3).

Then we prove that Dast's failovers preserve lemma 1. It suffices to prove three points. First, a newly added node will not assign timestamps smaller than an already-executed transaction. Dast ensures this by letting the newly added node to adjust its clock using a special CRT transaction (§4.4). Second, the newly added node can achieve a complete transaction history when executing a transaction T, which is ensured by letting all nodes set the `notifiedTs` entry for the newly added node to ts_{ckpt} (§4.4). Third, there is only one manager in each region at a time, and its timestamp is monotonic on failovers. The uniqueness of the manager is ensured by the SMR service [79], and the manager is recognized by all intra-region nodes using 2PC. The monotonicity of managers' `dclock` is ensured by letting newly elected managers collect the `maxTs` entry from all nodes (§4.4).

Proposition 1. *Dast ensures one-copy serializability.*

Proof sketch. Lemma 1 states that conflicting transactions execute in timestamp order at participating nodes, which suffices to prove serializability as the system's dependency graph is acyclic [89]. Lemma 1 also derives one-copy as replicas of a shard execute transactions in the same order.

Proposition 2. *For any two transactions T_1 and T_2 accessing the same region r, if T_2 starts after T_1 finishes, T_1 must precede T_2 in the serial order (i.e., $ts_2 > ts_1$).*

Proof sketch. We prove by contradiction. Suppose $ts_2 < ts_1$. As executing T_1 needs the `dclock` of all nodes in r to pass ts_1, T_2's participating nodes must have executed T_2, contradicting to the fact the T_2 starts after T_1 finishes.

Proposition 2 derives two important guarantees of Dast. First, Dast does not have stale reads as a data shard is only replicated on intra-region nodes. Second, the only possible anomalies in Dast is breaking the out-of-band dependencies of two transactions accessing different regions. In sum, Dast ensures one-copy serializability and no stale reads (§3.1).

4.6 Performance Analysis (R1, R2, and R3)

R1. In Dast, an IRT is not blocked by a CRT waiting for cross-region messages as long as intra-region RTT is much smaller than cross-region RTT, and nodes' system clocks are non-faulty (i.e., will not randomly advance tens of milliseconds). During intra-region network spikes, such blockings may occur, but these scenarios are rare because intra-region networks are usually connected with ultra-reliable and low-latency communication (URLLC) and LAN [54, 85, 107]. Moreover, during intra-region spikes, IRT latency is dominated by the intra-region RTT, and avoiding IRTs being blocked cannot effectively reduce their tail-latency.

Specifically, an IRT T_i may be blocked by a CRT T_c waiting for cross-region messages in two cases: (1) T_c is prepared and waiting for the `commit-crt` message, or (2) T_c is committed and waiting for remote-region inputs. In case (1), suppose T_i is assigned ts_i by its coordinator n^r, and T_c's prepared timestamp is ts_c^r. A necessary condition for blocking is $ts_i > ts_c^r$, which implies that when n^r receives T_i, its `dclock` passes ts_c^r, and it is unaware of the existence of T_c (with ts_c^r).

However, this condition is rare because intra-region nodes' `dlcocks` are calibrated by Dast's `dclock` calibration mechanism (§4.2) and when m^r anticipates ts_c^r, ts_c^r is one cross-region RTT ahead of its `dclock` value. As the cross-region RTT (e.g., hundreds of milliseconds) is usually much larger than intra-region RTT (e.g., less than ten milliseconds), n^r can usually receive the `prepare-crt` or notification message from other nodes (§4.3) before its `dclock` passes ts_c^r, even on retransmission on spontaneous packet losses. Case (2) is even rarer because the removal of ts_c^r and insertion of ts_c to a node's `waitQ` is atomic (§4.3), so n^r cannot assign $ts_i > ts_c$ if it cannot assign $ts_i > ts_c^r$.

R2. In Dast, a CRT is never aborted (except for conditional aborts) providing that no failover process is triggered. Consider a CRT T_c with commit timestamp ts_c. At any of its participating node n^r, Dast ensures that n^r will not execute any transactions with timestamps larger than ts_c before receiving the `commit-crt` message for T_c, even if n^r is not in the quorum for sending `ACK` for T_c's `prepare-crt` message. If n^r is to execute transactions with timestamps larger than ts_c, Dast requires n^r to be notified that all intra-region nodes' `dclocks` have passed ts_c (§4.3), and n^r will get notified the existence of T_c in such notifications. Therefore, on any participating node n^r, T_c does not need to be aborted.

R3. In Dast, committing a CRT only involves participating regions, without going through a global centralized service. As shown in §6.2, a global centralized service for all CRTs can easily become a scalability bottleneck.

5 Implementation

We implemented Dast with 3011 lines of C++ code on the Janus codebase [6], a modular framework for evaluating distributed databases. All of Dast's protocol messages are implemented with asynchronous RPC calls. Each node has two threads, one for checking and executing transactions, and the other one for handling I/O requests in the RPC server.

We support cross-region value dependencies by analyzing the dependencies among pieces when a transaction is submitted. Specifically, each input or output variable has a `varId`, and each transaction contains a map of `⟨varId, shardId⟩` in its metadata, recording the dependent shard (if any) of each variable. When a piece generates an output with a variable in the map, the output is sent to replicas of the target shard with a `SendOutput` RPC call.

6 Evaluation

All experiments except for the scalability experiment were done on our cluster with 25 machines, each with a 2.60GHz Intel E5-2690 CPU with 24 cores, 40Gbps NIC, and 64GB memory. The scalability experiment was done on AWS with up to 100 c5.18xlarge instances in the US East (Ohio) region.

txn	**new-order**		**payment**		**order-status**		**delivery**		**stock-level**	
type	IRT	CRT	IRT	CRT	IRT	CRT	IRT	CRT	IRT	CRT
ratio	39.60%	4.38%	37.51%	6.57%	3.96%	0%	3.95%	0%	4.02%	0%
total	43.98%		44.08%		3.96%		3.95%		4.02%	

Table 2. Transaction mix ratio of TPC-C in a Dast trial.

We ran each node in a docker container and used tc [12] to control the RTT among nodes.

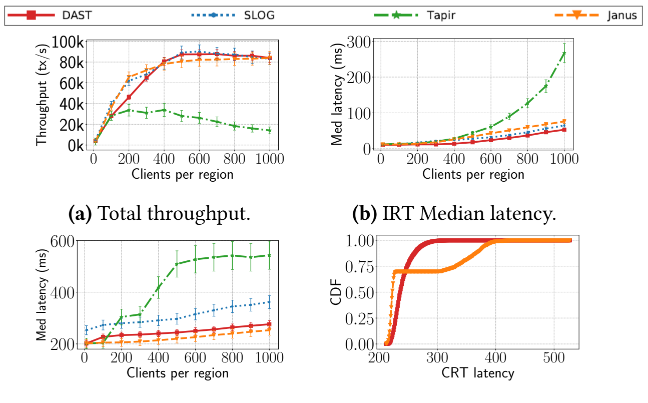
Baseline. We compared Dast with three latest serializable geo-distributed databases: SLOG [86], Janus [73], and Tapir [113]. We chose them because Janus is our codebase; Tapir is a state-of-the-art deferred update database; SLOG is a latest SMR-based database that exploits transactions' spatial locality. Their protocols are described in §2. We did not compare to other edge databases because they either do not support multi-shard transactions (e.g., DPaxos [75]) or provide weaker guarantees than serializability (e.g., SEQ [59]).

For an apple-to-apple comparison, all baseline systems were implemented on the Janus framework [4]. We extended SLOG and Tapir to have the same guarantee §3.1 as Dast. For Tapir, we extended the implementation evaluated and calibrated in the Janus paper [73] to a non-strict serializable version presented in its paper [113]. We implemented SLOG with 2452 lines of code, using Raft [79] with three replicas as its global ordering service, as suggested in its paper [86]. We set the log exchange interval to 5ms, same as its open-source code [5]. To make SLOG have the same guarantee as Dast, we let a shard release a transaction T's lock once the execution of T's pieces accessing this shard finishes (i.e., downgrading from strong strict two-phase locking to two-phase locking [23]); We did not use SLOG's code because it does not support stored procedures.

Overall, our evaluated SLOG and Tapir implementations have the same serializability guarantee (§3.1) as Dast, and Janus ensures strict serializability.

Workloads. As far as we know, there is no standard benchmark for serializable edge databases. Therefore, for a fair comparison, we evaluated workloads with locality features in all relevant conventional databases [27, 36, 46, 73, 76, 86, 99, 109, 113], including TPC-C-default (evaluated in Janus [73], SLOG [86], Calvin [99], Spanner [27]), TPC-A (a comparable workload to YCSB, as YCSB was used in Tapir [113] and Carousel [109]), and TPC-C Payment-only. Specifically, we used TPC-C Payment-only to stress-test Dast on different portions of CRTs because the TPC-C Payment transaction is read-write intensive and has cross-region value dependencies. Although retwis [52] was evaluated in Carousel [109] and Tapir [113], we did not evaluate retwis because it does not have locality feature and thus is not edge-relevant.

Although these benchmarks are general and not dedicated to edge scenarios, they have already covered diverse data access patterns in various edge-computing applications. For instance, edge applications often exhibit good spatial locality (i.e., most of a client's transactions access data held by nearby



(a) Total throughput. **(b)** IRT Median latency.

(c) CRT Median latency. **(d)** CRT CDF of Dast and Janus.

Figure 5. TPC-C performance with different numbers of clients.

nodes), which have been well emulated by TPC-C, since around 90% of the client's transactions access data in its own warehouse [14]. The TPC-C payment transaction reads an account and writes to a contended data field, similar to the trade booking transaction in edge-based high-frequency and latency-sensitive trading [115]. The TPC-A benchmark with different conflict rates is similar to charging plant scheduling in areas of different user densities.

Deployment. To match edge deployments, we horizontally partitioned the TPC-C database based on warehouse id (i.e., each shard is a warehouse). We ran 10∼100 regions, each containing 30 edge nodes holding 10 shards with a replication level of 3. Since a TPC-C client is already associated with a warehouse ID, we assigned each client to the region containing the client's warehouse. Each client sent transactions in closed-loop to a random replica of its warehouse. As in recent edge computing surveys [91], we set the intra-region RTT (between two nodes or from a client to a node) to 5ms and the cross-region RTT to 100ms. We did not run our experiments across AWS regions [11] because AWS does not have enough regions (i.e., 100). We set the same RTT among regions by default to ease the analysis and discussions of the tail latency of each system.

We ran each experiment for 30 seconds and collected the result in the middle 15s (i.e., 7.5s∼22.5s) to avoid the disturbance caused by system start-up and cool-down. For latency, we measured the client-side latency, including retries. For tail latency, we measured the 99-percentile tail latency to match the requirements of mission-critical applications on edge nodes [85, 107]. We did not use 95-percentile latency because TPC-C has transaction types with a mix ratio smaller than 5%.

Our evaluation focused on these questions:

§6.1 : How do Dast's throughput and latency compare to relevant serializable conventional databases?
§6.2 : Can these systems scale to a large number of regions?
§6.3 : Is Dast sensitive to network anomalies?
§6.4 : What are the limitations and potentials of Dast?

Phase	**remote prepare**	**local prepare**	**wait exe.**	**wait input**	**wait output**	**total**
avg. time w/o dep (ms)	107.8	7.3	13.3	0	87.6	216.3
avg. time w/ dep (ms)	107.1	7.2	14.9	86.7	1.4	218.4

Table 3. Latency breakdown of CRTs in Dast on default TPC-C with 500 clients per region (peak throughput). "w/ dep" & "w/o dep" represent whether a transaction has cross-region value dependencies; "remote prepare" is the time for collecting anticipated timestamp from participating regions; "local prepare" is for collecting replicating anticipated timestamp within the coordinator's region; "wait exe." is the waiting time in the `readyQ`; "wait input" is for waiting for input values from remote regions; "wait output" is for waiting for transaction output from remote regions.

6.1 Performance on Different Workloads

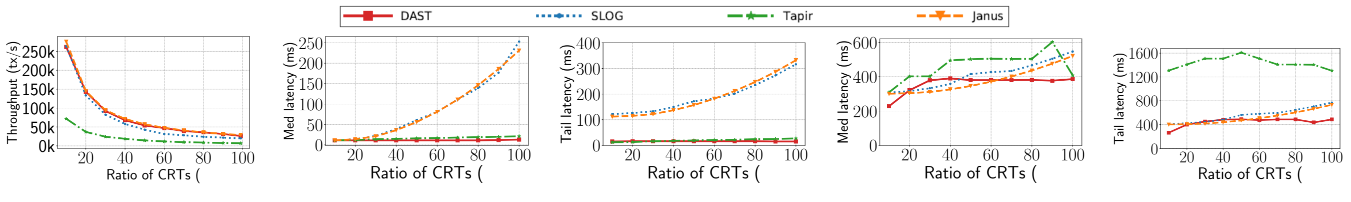
TPC-C-default. Figure 2 and Figure 5 show the performance of the four systems. Overall, Dast was the only system meeting both **R1** and **R2**. Dast reduced the tail latency by 87.9%∼93.2% for IRTs and 27.7%∼70.4% for CRTs compared to three relevant systems and achieved comparable throughput. Dast's CRT median latency was 7.6% higher than Janus on average; we consider this tradeoff worthwhile considering the reduction of tail latency for both IRTs and CRTs. Each node consumed at most 40.8 Mbps bandwidth, which can be fulfilled by existing edge data centers [19, 114].

For throughput, Dast, Janus, and SLOG climbed until their CPUs were saturated. SLOG dropped slightly after 600 clients per region due to the large number (up to 1000×10) of concurrent connections to the global ordering server. Tapir's throughput dropped due to aborts and retries [36, 73, 109].

The IRT median latency of Dast, SLOG, and Janus increased slightly due to the queuing time after each system was saturated. Tapir's median latency increased dramatically because IRTs were aborted and retried multiple times.

The IRT tail latency for SLOG and Janus increased from 1RTT because they order conflicting transactions using the FCFS approach (§2), resulting in IRTs blocked by CRTs. We looked into the logs and found that such blockings were mainly due to CRTs of type `payment-by-name`, where a cross-region value dependency is needed for retrieving client ID. Therefore, we chose TPC-C-payment only as our stress test workload. We did not observe such blockings in Dast, and Dast's IRT tail latency increased due to the queuing time. Tapir's IRT tail latency was low when the number of clients was small because IRTs are not blocked (§2), but increased dramatically when the conflict rate increased.

Dast's CRT median latency was higher than Janus's because Dast prioritizes IRTs. To further understand Dast's CRT latency, we collected its breakdown, shown in Table 3. The "wait for execution" part was faster than half RTT because Dast's `dclock` calibration protocol makes the clock go faster (§4.2). However, the `commit` message and transaction outputs took one RTT to travel back, and transactions with or without cross-region value dependencies wait for



(a) Total throughput. **(b)** IRT median latency. **(c)** IRT 99% tail latency. **(d)** CRT median latency. **(e)** CRT 99% tail latency.

Figure 6. Performance on different CRT ratios in the TPC-C Payment-only workload. We selected the number of clients that made each system reach its peak throughput on the most heavily loaded 1% trial (error bars omitted for readability).

Phase	remote prepare	local prepare	wait exe.	wait input	wait output	total
avg. time w/o dep (ms)	107.4	7.3	246.2	0	18.4	379.3
avg. time w/ dep (ms)	108.1	7.2	235.7	78.4	7.03	446.8

Table 4. Latency breakdown of CRTs in Dast for the payment-only workload with 40% of all transactions being CRTs.

this RTT in different phases of their transaction. In either case, the waiting did not block subsequent IRTs thanks to Dast's stretchable clock (§4.3).

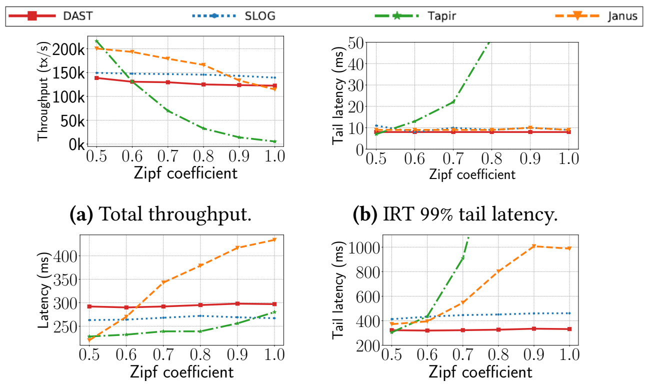
To understand why Dast's CRT median latency was higher than Janus's while CRT tail latency was lower, we collected their CRT CDF with 500 clients per region (peak throughput), shown in Figure 5d. Janus's CRT median latency was around 2 RTTs because more than half of the CRTs entered its fast path, which used 1 RTT to commit a CRT and 1 RTT for collecting outputs. Its tail latency was around 4 RTTs, instead of 3 RTTs in its slow path protocol (§2). We looked into its log and found the additional RTT is caused by a dependent piece of one CRT blocked by other CRTs' pieces waiting for inputs.

SLOG's CRT median latency climbed from about 2.5 RTTs and increased faster than Dast and Janus due to the queuing effect at the global ordering service. Tapir's median and tail latency for CRTs increased dramatically because CRTs were retried many times.

Performance on different ratios of CRTs. To stress-test Dast's performance on different ratios of CRTs, we selected the `payment` transaction and adjusted its CRT ratio from 1% to 99%. Around 60% of the CRTs have cross-region value dependencies [14]. The result is shown in Figure 6. All systems' throughput dropped when the ratio of CRTs increased because a CRT's finish time is much longer than an IRT's.

Dast's median and tail IRT latency was stable because IRTs were never blocked by CRTs. Tapir's latency was also stable because the conflict rate was low with the number of clients making Tapir reach its peak throughput. Janus's and SLOG's IRT latency increased with the CRT ratio as the probability that an IRT is blocked by CRTs increased.

Dast's CRT latency (median and tail) increased when the CRT ratio increased from 0% to 40%. To understand the reason, we collected the breakdown of CRT latency with 40% CRT ratio (i.e., when Dast achieved the highest latency), shown in Table 4. Comparing it with Table 3, the major increment was from the "waiting for execution" phase. This is because a node's `dclock` froze its `time` field when a CRT is



(a) Total throughput. **(b)** IRT 99% tail latency. **(c)** CRT median latency. **(d)** CRT 99% tail latency.

Figure 7. Performance on different conflict rates in the TPC-A workload (error bars omitted for readability).

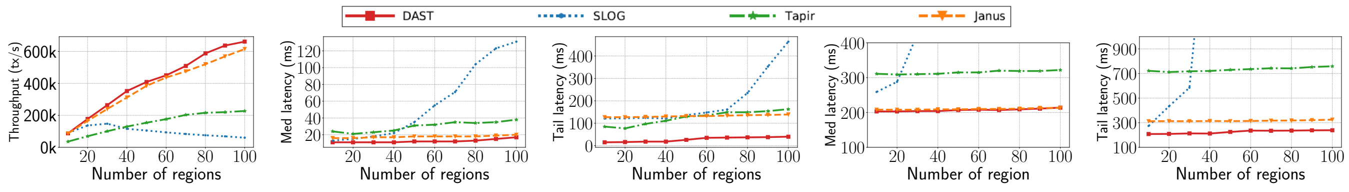
waiting for its input to prioritize IRTs, which delays subsequent CRTs. The increment stalled after the CRT ratio was more than 40% because the total throughput dropped.

The CRT latency (median and tail) of Janus and SLOG increased with the CRT ratio because CRTs with cross-region dependencies blocked each other and accumulated. Note that such blockings only happen in edge deployments and will not happen if the database is fully-replicated among a few conventional data centers. Tapir's conflict rate first increased because IRTs had fewer conflicts than CRTs due to shorter finish time, and then dropped due to the drop in system load.

Overall, Dast ensures low latency for IRTs (**R1**) regardless of CRT ratios, but Dast may achieve a high latency for CRTs if the CRT ratio is high (e.g., larger than 30%). Dast is complementary to conventional databases: Dast is more suitable for typical edge workloads [35, 59, 75, 88] with good locality features, while workloads without locality features are more suitable to be served by conventional databases.

Performance on different conflict rates. We ran the TPC-A micro-benchmark and changed the zipf coefficients from 0.5 to 1.0. The result is shown in Figure 7. Overall, Dast is insensitive to conflict rates because Dast relies on the intrinsic orders of unique timestamps to order all transactions.

For throughput, when the conflict rate was low, Janus achieved the highest throughput because most transactions entered the fast path. SLOG's throughput was also higher than Dast thanks to the batching of small transactions.



(a) Total throughput. **(b)** IRT median latency. **(c)** IRT 99% tail latency. **(d)** CRT median latency. **(e)** CRT 99% tail latency.

Figure 8. Performance on different numbers of regions in the TPC-C workload (error bars omitted for readability).

DAST's throughput was relatively stable because DAST orders all transactions regardless of whether they conflict.

All systems' IRT latency was stable except for Tapir. This is because TPC-A has only independent transactions without cross-region value dependencies. Tapir's latency increased when the conflict rate was high.

The CRT latency of DAST and SLOG was stable because they order all transactions regardless of whether they conflict. Tapir's latency increased due to conflicts, consistent with the result reported in Janus's paper [73]. Janus's latency increased due to two reasons. First, Janus switched from its fast path to its slow path when the conflict rate increased. Second, a node needed to send an `inquire` message to know whether a dependent transaction had been committed (§2).

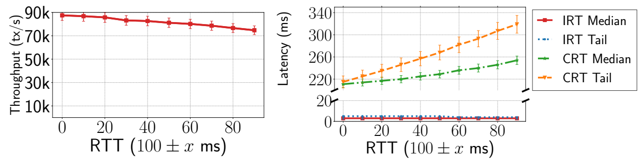
6.2 Scalability to the Number of Regions

We ran 10 to 100 regions and let each region have the number of clients that made each system achieve peak throughput in Figure 5. The result is shown in Figure 8. DAST, Janus, and Tapir showed good scalability: their throughput scaled near linearly to the number of regions. This is because in these three systems, IRTs in each region work independently, and IRTs contributed to around 90% (Table 2) of total throughput.

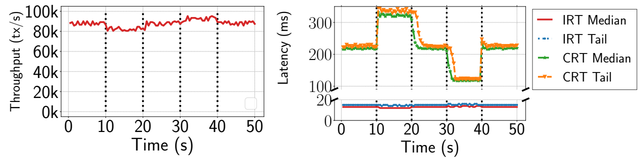
To further understand DAST's performance, we also looked into its CRT throughput per region and found that this number is roughly stable, changing from 953 for 10 regions to 866 for 100 regions. The drop was mainly due to the increased number of RPC connections on each manager. This result is as expected because in TPC-C, the number of regions accessed by each CRT does not increase with the total number of regions, so the number of CRTs accessing each region does not change with the total number of regions. This also explains why the latency of these three systems was stable.

SLOG's throughput increased from 10 to 30 regions and started to drop because the global ordering service is a scalability bottleneck. When the number of regions was 100, the leader needs to handle CRTs from 50000 clients and to send *each* CRT to all regions' managers. We investigated into the log and found that time for dispatching a CRT increased from 685us to 5.48ms on average. Therefore, CRT latency increased dramatically, and as the clients were in close-loop, the system's throughput dropped accordingly.

Overall, DAST, Tapir, and Janus meet **R3**, and DAST is the only system that meets all three requirements simultaneously. Moreover, the result also indicates that DAST has the potential to scale to a large number of regions on typical edge workloads, because DAST scales near linearly to the number of regions, and the only bottleneck of marinating concurrent RPC connections is from the implementation level, which has been actively explored in both academia [40, 93, 100, 112] and industry [1, 2, 8]. Note that SLOG's scalability bottleneck is that its ordering server needs to send CRTs to a large number of servers, instead of simply maintaining connections.



(a) Throughput on fluctuation. **(b)** Latency on fluctuation.



(c) Throughput on spikes. **(d)** Latency on spikes.

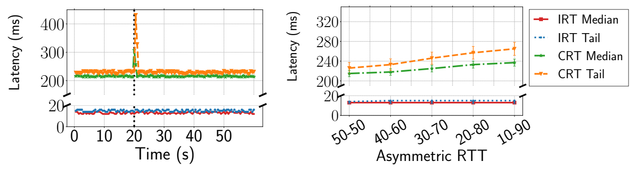
Figure 9. DAST's performance on cross-region network anomalies.

6.3 Robustness to Network Anomalies

We first set the cross-region RTTs to 100 ± x ms with a uniform distribution. We spawned clients with the number that made DAST reach peak throughput in Figure 5. As shown in Figure 9b, DAST's latency for IRTs was stable (**R1**) because DAST's hybrid clock avoids a CRT from blocking IRTs even if the anticipation of CRT's commit time is inaccurate (§4.2). The latency of CRTs increased roughly proportional with x increasing, which infers that DAST's performance for CRTs is robust because the disturbance of network anomalies did not accumulate. DAST's throughput showed only a little drop because IRTs contributed to most of the throughput.

We then evaluated DAST's performance on abrupt changes of cross-region RTT (e.g., caused by network spikes). We changed the RTT among regions every 10s (vertical lines in the figures): we increased the cross-region RTT from 100ms to 150ms at 10s and back to 100ms at 20s; then, we changed the RTT from 100ms to 50ms at 30s and back to 100ms at 40s. We collected the latency and throughput every 500ms.

The IRT latency was stable because an IRT is not blocked by CRTs. The slight change was due to the change in system load as we selected the number of clients that made DAST just saturate. The CRT latency dropped slowly when the



(a) Latency on clock skewness (skewness added at 20s). **(b)** Latency on clock skewness and asymmetric network latency.

Figure 10. DAST's performance on cross-region clock skewness.

RTT dropped because DAST uses the average RTT based communication history (§4.3) to estimate RTTs among regions; when the network latency dropped, the average RTT is larger than the real RTT until the average RTT dropped. Nevertheless, inaccurate estimation of CRT does not block subsequent IRTs. Overall, DAST is robust to network anomalies as DAST's hybrid clock ensures that IRTs are not blocked by CRTs (**R1**) regardless of cross-region network anomalies.

To evaluate DAST's performance on cross-region clock skewness, we ran two regions, each with 10 shards and 3 replicas per shard. At 20s, we advanced the system clock of the manager node in the second region by 200ms and shut down its NTP process. Figure 10a shows the result (collected every 500ms). DAST's IRT latency was stable, but CRT latency showed an obvious spike because these CRTs were assigned a large anticipated timestamp when we advance the clock. The increment of CRT median latency was slower because the CRTs coordinated by the region whose clocks are faster were not obviously affected. The latency drops soon because DAST's clock calibration mechanism (§4.2 and §4.3) advanced the `dclcoks` for other nodes to catch up.

To evaluate DAST's performance in the presence of both clock skewness and asymmetric network delay, we ran DAST with two regions, each with 10 shards and 3 replicas per shard. We set the clocks for nodes in the second region to be 200ms faster than nodes in the first region and disabled NTP. The RTT was still 100ms but we altered the one-way delay between the two regions. As shown in Figure 10b, DAST's CRT latency increased as the asymmetry became severe because DAST's optimization assumes a symmetric network (§4.3). However, in most real-world networks, one-way delay takes no more than 70% of the total RTT [81].

6.4 Discussion

DAST has four limitations. First, same as existing partially replicated database systems (e.g., CockroachDB [67]), DAST ensures one-copy serializability instead of strict serializability. However, if developers explicitly write down out-of-band dependencies among transactions accessing non-intersecting regions, DAST will have the same guarantee as strict serializable databases (§4.5). Second, DAST supports only workloads written in stored procedures. Stored procedures are widely used in distributed databases [37, 43, 72, 73, 95, 101] due to its good performance, maintainability, and security [39, 80, 105], and all workloads evaluated by relevant databases [13, 14, 30, 52] can be written as stored procedures. Third, DAST requires a CRT's value dependencies to be acyclic (§4.1). Nevertheless, DAST's transaction model is already more general than the widely used independent transaction model [28, 41, 43, 73] that prohibits all value dependencies. As noted in [98, 109], value dependencies are usually used for accessing values by secondary index in real-world workloads, so cyclic dependencies are usually rare [72]. In the presence of such CRTs, DAST can easily detect them by using the simple analysis mechanism (§5), and DAST can support them either by letting CRTs abort on conflicts to ensure low latency for IRTs or by letting IRTs wait. We plan to leave this as our future work. Fourth, DAST targets workloads [35, 59, 75, 88] with good locality, while workloads without such a locality feature are more suitable to be served by geo-distributed databases [27, 36, 46, 73, 76, 86, 99, 109, 113] in conventional data centers.

Overall, DAST is complementary to conventional databases: when the CRTs take just a minor portion of all transactions, DAST enforces much lower, stable, and scalable tail latency for both IRTs and CRTs than conventional databases; otherwise, conventional databases could be more suitable (see §6.1). DAST is also complementary to existing edge databases [59, 75]: DAST provides serializable data access to mission-critical applications while existing edge databases can provide faster data accesses for applications that require only weak consistency (e.g., web-browsing). DAST has the potential to facilitate the porting of various applications to the edge to harness the locality of clients' data access and to provide better user experience.

7 Conclusion

We presented the design, implementation, and evaluation of DAST, the first distributed edge database that enforces both serializability and the three crucial performance requirements to support mission-critical applications deployed on edge computing nodes. DAST schedules transactions based on anticipating when they are ready to execute, and DAST carries a new stretchable clock abstraction to tolerate inaccurate anticipations in the asynchronous Internet. DAST's code is released on github.com/hku-systems/dast.

Acknowledgments

We thank our shepherd, Luis Rodrigues, and all anonymous reviewers for their helpful comments. This work is funded by the research grants from Huawei Innovation Research Program Flagship 2018, HK RGC GRF (17202318, 17207117), HK RGC ECS (27200916), National Nature Science Foundation of China (61802358), RGC Projects HKU (17229116 and 106150091), University of Hong Kong (Projects 104005858 and 104005994), the Guangdong-Hong Kong-Macau Joint Laboratory Program 2020 (2020B1212030009), and a Croucher Innovation Award.

References

[1] Apache Thrift. http://thrift.apache.org/.

[2] apache/incubator-brpc. https://github.com/apache/incubator-brpc.

[3] Azure IoT Edge. https://azure.microsoft.com/en-gb/services/iot-edge/.

[4] Github: a stable commit of NYU-NEWS/Janus. https://github.com/NYU-NEWS/janus/tree/3c1b60cd965551b4bd3b1f3c475e16fdde2e4c2b.

[5] Github: ctring/SLOG. https://github.com/ctring/slog.

[6] Github: NYU-NEWS/Janus. https://github.com/NYU-NEWS/janus.

[7] Google Cloud IoT Core. https://cloud.google.com/iot-core/.

[8] gRPC: A high-performance, open source universial RPC framework. https://grpc.io/.

[9] How alaska outsmarts mother nature in the cloud. https://customers.microsoft.com/en-us/story/alaskadotpf-government-azure-iot.

[10] Internet of aircraft things: An industry set to be transformed. https://aviationweek.com/connected-aerospace/internet-aircraft-things-industry-set-betransformed.

[11] Regions, Availability Zones, and Local Zones. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

[12] tc(8), linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html.

[13] TPC-A. http://www.tpc.org/tpca/, 2014.

[14] TPC-C. http://www.tpc.org/tpcc/, 2014.

[15] Daniel Abadi. Correctness anomalies under serializable isolation. http://dbmsmusings.blogspot.com/2019/06/correctness-anomalies-under.html.

[16] Daniel Abadi and Matt Freels. Serializability vs strict serializability: The dirty secret of database isolation levels. https://fauna.com/blog/serializability-vs-strict-serializability-the-dirty\-secret-of-database-isolation-levels.

[17] Nuno Afonso, Manuel Bravo, and Luís Rodrigues. Combining high throughput and low migration latency for consistent data storage on the edge. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–11. IEEE, 2020.

[18] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. Wpaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):211–223, 2019.

[19] Carlos Andrés Ramiro, Claudio Fiandrino, Alejandro Blanco Pizarro, Pablo Jimenez Mateo, Norbert Ludant, and Joerg Widmer. openleon: An end-to-end emulator from the edge data center to the mobile users. In *Proceedings of the 12th International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization*, pages 19–27, 2018.

[20] Gargi Bag, Linus Thrybom, and Petri Hovila. Challenges and opportunities of 5g in power grids. *CIRED-Open Access Proceedings Journal*, 2017(1):2145–2148, 2017.

[21] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. *ACM SIGMOD Record*, 24(2):1–10, 1995.

[22] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley Reading, 1987.

[23] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, (3):203–216, 1979.

[24] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, October 1999.

[25] Jiangfeng Cheng, Weihai Chen, Fei Tao, and Chun-Liang Lin. Industrial iot in 5g environment towards smart manufacturing. *Journal of Industrial Information Integration*, 10:10–19, 2018.

[26] Franco Cicirelli, Antonio Guerrieri, Giandomenico Spezzano, and Andrea Vinci. An edge-based platform for dynamic smart city applications. *Future Generation Computer Systems*, 76:106–118, 2017.

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

[28] James Cowling and Barbara Liskov. Granola: low-overhead distributed transaction coordination. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 223–235, 2012.

[29] Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. *IEEE Transactions on industrial informatics*, 10(4):2233–2243, 2014.

[30] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. Ycsb+t: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 223–230. IEEE, 2014.

[31] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *Proceedings of the VLDB Endowment*, 12(2):169–182, 2018.

[32] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone. Clock-rsm: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 343–354, 2014.

[33] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, pages 173–184. IEEE, 2013.

[34] Dominik Durner and Thomas Neumann. No false negatives: Accepting all useful schedules in a fast serializable many-core system. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 734–745. IEEE, 2019.

[35] Ittay Eyal, Ken Birman, and Robbert van Renesse. Cache serializability: Reducing inconsistency in edge transactions. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 686–695. IEEE, 2015.

[36] Hua Fan and Wojciech Golab. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. volume 12, pages 1471–1484. VLDB Endowment, 2019.

[37] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering*, 4(6):509–516, 1992.

[38] Chathuri Gunawardhana, Manuel Bravo, and Luis Rodrigues. Unobtrusive deferred update stabilization for efficient geo-replication. In *2017 USENIX Annual Technical Conference (USENIXATC 17)*, pages 83–95, 2017.

[39] Er Shobhit Gupta et al. Database management-inline queries vs. stored procedures–an extended analysis. 2015.

[40] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, 2014.

[41] Li Jialin, Michael Ellis, and Ports Dan. Eris: Coordination-free consistent transactions using in-network concurrency control. In *SOSP*, 2017.

[42] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Bettina Kemme, and Gustavo Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 477–484. IEEE, 2002.

[43] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[44] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*, pages 1675–1687, San Francisco, California, USA, 2016. ACM Press.

[45] Mustafa Korkmaz, Martin Karsten, Kenneth Salem, and Semih Salihoglu. Workload-aware cpu performance scaling for transactional database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 291–306, 2018.

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

[47] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *International Conference on Principles of Distributed Systems*, pages 17–32. Springer, 2014.

[48] Hsiang-Tsung Kung and John T Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[49] Leslie Lamport. Paxos made simple. http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf.

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

[51] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[52] Costin Leau. Spring Data Redis - Retwis-J. https://https://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[53] Liang Li, Yunzhou Li, and Ronghui Hou. A novel mobile edge computing-based architecture for future cellular vehicular networks. In *2017 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2017.

[54] Zexian Li, Mikko A Uusitalo, Hamidreza Shariatmadari, and Bikramjit Singh. 5g urllc: Design challenges and system concepts. In *2018 15th International Symposium on Wireless Communication Systems (ISWCS)*, pages 1–6. IEEE, 2018.

[55] Zhongmiao Li, Paolo Romano, and Peter Van Roy. Sparkle: Speculative Deterministic Concurrency Control for Partially Replicated Transactional Stores. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 164–175, Portland, OR, USA, June 2019. IEEE.

[56] Zhongmiao Li, Peter Van Roy, and Paolo Romano. Speculative transaction processing in geo-replicated data stores. Technical report, Technical Report 2, INESC-ID, 2017.

[57] Chun-Cheng Lin and Jhih-Wun Yang. Cost-efficient deployment of fog computing systems at logistics centers in industry 4.0. *IEEE Transactions on Industrial Informatics*, 14(10):4603–4611, 2018.

[58] Guoqiang Lin, Siyan Liu, Aihua Zhou, Jiangpeng Dai, Bo Chai, Bo Zhang, Hongbin Qiu, Kunlun Gao, Yan Song, and Rui Chen. Community detection in power grids based on louvain heuristic algorithm. In *2017 IEEE Conference on Energy Internet and Energy System Integration (EI2)*, pages 1–4. IEEE, 2017.

[59] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. Enhancing edge computing with database replication. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 45–54. IEEE, 2007.

[60] Jianqi Liu, Jiafu Wan, Bi Zeng, Qinruo Wang, Houbing Song, and Meikang Qiu. A scalable and quick-response software defined vehicular network assisted by mobile edge computing. *IEEE Communications Magazine*, 55(7):94–100, 2017.

[61] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[62] Dumitrel Loghin, Shaofeng Cai, Gang Chen, Tien Tuan Anh Dinh, Feiyi Fan, Qian Lin, Janice Ng, Beng Chin Ooi, Xutao Sun, Quang-Trung Ta, et al. The disruptions of 5g on data-driven technologies and applications. *arXiv preprint arXiv:1909.08096*, 2019.

[63] Yi Lu, Xiangyao Yu, and Samuel Madden. Star: Scaling transactions through asymmetric replication. *Proceedings of the VLDB Endowment*, 12(11).

[64] Gang Luo, Jeffrey F Naughton, Curt J Ellmann, and Michael W Watzke. Transaction reordering with application to synchronized scans. In *Proceedings of the ACM 11th international workshop on Data warehousing and OLAP*, pages 17–24, 2008.

[65] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. MaaT: effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.*, 7(5):329–340, January 2014.

[66] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, volume 8, pages 369–384, 2008.

[67] Andrei Matei. Cockroachdb's consistency model. https://www.cockroachlabs.com/blog/consistency-model/, 2019.

[68] David Mazieres. Paxos made practical. Technical report, Technical report, 2007. http://www. scs. stanford. edu/dm/home/papers, 2007.

[69] Hein Meling, Keith Marzullo, and Alessandro Mei. When you don't trust clients: Byzantine proposer fast paxos. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, 2012.

[70] D.L. Mills. IEN173:Time Synchronization in DCNET Hosts. https://web.archive.org/web/19961230073104/http://www.cis.ohio-state.edu/htbin/ien/ien173.html, 1981.

[71] Hassnaa Moustafa and Yan Zhang. *Vehicular networks: techniques, standards, and applications.* Auerbach publications, 2009.

[72] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 479–494, Broomfield, CO, October 2014. USENIX Association.

[73] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 517–532, 2016.

[74] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9(4):680–710, 1984.

[75] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1221–1236, 2018.

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

[77] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15*, pages 1279–1294,

Melbourne, Victoria, Australia, 2015. ACM Press.

[78] Zhaolong Ning, Peiran Dong, Xiaojie Wang, Joel JPC Rodrigues, and Feng Xia. Deep reinforcement learning for vehicular edge computing: An intelligent offloading system. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(6):1–24, 2019.

[79] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC '14)*, June 2014.

[80] Orcale. Advantages of Stored Procedures. https://docs.oracle.com/cd/F49540_01/DOC/java.815/a64686/01_intr3.htm.

[81] Abhinav Pathak, Himabindu Pucha, Ying Zhang, Y Charlie Hu, and Z Morley Mao. A measurement study of internet delay asymmetry. In *International Conference on Passive and Active Network Measurement*, pages 182–191. Springer, 2008.

[82] Al-Mukaddim Khan Pathan and Rajkumar Buyya. A taxonomy and survey of content delivery networks. *Grid Computing and Distributed Systems Laboratory, University of Melbourne, Technical Report*, 4:70, 2007.

[83] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[84] Michael George Polan. Method and apparatus for presenting navigable data center information in virtual reality using leading edge rendering engines, March 17 2009. US Patent 7,506,264.

[85] Tie Qiu, Jiancheng Chi, Xiaobo Zhou, Zhaolong Ning, Mohammed Atiquzzaman, and Dapeng Oliver Wu. Edge computing in industrial internet of things: Architecture, advances and challenges. *IEEE Communications Surveys & Tutorials*, 22(4):2462–2488, 2020.

[86] Kun Ren, Dennis Li, and Daniel J Abadi. Slog: serializable, low-latency, geo-replicated transactions. volume 12, pages 1747–1761. VLDB Endowment, 2019.

[87] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[88] Hemant Saxena and Kenneth Salem. EdgeX: Edge Replication for Web Applications. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 1041–1044, June 2015. ISSN: 2159-6190.

[89] Marc H Scholl. Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. *ACM SIGMOD Record*, 30(4):67–68, 2001.

[90] Zechao Shang. Next generation consistency enforcement. In *CIDR*, 2017.

[91] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[92] Ankit Singla, Balakrishnan Chandrasekaran, P Brighten Godfrey, and Bruce Maggs. The internet at the speed of light. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2014.

[93] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.

[94] António Sousa, José Pereira, Francisco Moura, and Rui Oliveira. Optimistic total order in wide area networks. In *21st IEEE Symposium on Reliable Distributed Systems, 2002. Proceedings.*, pages 190–199. IEEE, 2002.

[95] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: It's time for a complete rewrite. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 463–489. 2018.

[96] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[97] Fei Tao, Ying Zuo, Li Da Xu, and Lin Zhang. Iot-based intelligent perception and access of manufacturing resource toward cloud manufacturing. *IEEE Transactions on Industrial Informatics*, 10(2):1547–1557, 2014.

[98] Alexander Thomson and Daniel J Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.

[99] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Fast distributed transactions and strongly consistent replication for oltp database systems. May 2014.

[100] Supachai Thongprasit, Vasaka Visoottiviseth, and Ryousei Takano. Toward fast and scalable key-value stores based on user space tcp/ip stack. In *Proceedings of the Asian Internet Engineering Conference*, pages 40–47, 2015.

[101] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[102] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. *ACM SIGOPS Operating Systems Review*, 41(6):59–72, 2007.

[103] Kai Wang, Hao Yin, Wei Quan, and Geyong Min. Enabling collaborative edge computing for software defined vehicular networks. *IEEE Network*, 32(5):112–117, 2018.

[104] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment*, 10(2):49–60, 2016.

[105] Kei Wei, Muthusrinivasan Muthuprasanna, and Suraj Kothari. Preventing sql injection attacks in stored procedures. In *Australian Software Engineering Conference (ASWEC'06)*, pages 8–pp. IEEE, 2006.

[106] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, pages 206–215. IEEE, 2000.

[107] Hansong Xu, Wei Yu, David Griffith, and Nada Golmie. A survey on industrial internet of things: A cyber-physical systems perspective. *IEEE Access*, 6:78238–78259, 2018.

[108] Xinan Yan, Linguan Yang, and Bernard Wong. Domino: using network measurements to reduce state machine replication latency in wans. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 351–363, 2020.

[109] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*, pages 231–243, 2018.

[110] Shun-Ren Yang, Yu-Ju Su, Yao-Yuan Chang, and Hui-Nien Hung. Short-term traffic prediction for edge computing-enhanced autonomous and connected cars. *IEEE Transactions on Vehicular Technology*, 68(4):3140–3153, 2019.

[111] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16*, pages 1629–1642, San Francisco, California, USA, 2016. ACM Press.

[112] Matthew J Zelesko and David R Cheriton. Specializing object-oriented rpc for functionality and performance. In *Proceedings of 16th International Conference on Distributed Computing Systems*, pages 175–187. IEEE, 1996.

[113] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.*, 35(4):1–37, December 2018.

[114] Yuan Zhang. Edge cloud infrastructure for the future network.

[115] Zhenyu Zhou, Bingchen Wang, Mianxiong Dong, and Kaoru Ota. Secure and efficient vehicle-to-grid energy trading in cyber physical systems: Integration of blockchain and edge computing. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 50(1):43–57, 2019.