RT-ZooKeeper: Taming the Recovery Latency of a Coordination Service

HAORAN LI, CHENYANG LU, and CHRISTOPHER D. GILL, Cyber-Physical Systems Laboratory, Washington University in St. Louis, USA

Fault-tolerant coordination services have been widely used in distributed applications in cloud environments. Recent years have witnessed the emergence of time-sensitive applications deployed in edge computing environments, which introduces both challenges and opportunities for coordination services. On one hand, coordination services must recover from failures in a timely manner. On the other hand, edge computing employs local networked platforms that can be exploited to achieve timely recovery. In this work, we first identify the limitations of the leader election and recovery protocols underlying Apache ZooKeeper, the prevailing open-source coordination service. To reduce recovery latency from leader failures, we then design RT-Zookeeper with a set of novel features including a fast-convergence election protocol, a quorum channel notification mechanism, and a distributed epoch persistence protocol. We have implemented RT-Zookeeper based on ZooKeeper version 3.5.8. Empirical evaluation shows that RT-ZooKeeper achieves 91% reduction in maximum recovery latency in comparison to ZooKeeper. Furthermore, a case study demonstrates that fast failure recovery in RT-ZooKeeper can benefit a common messaging service like Kafka in terms of message latency.

CCS Concepts: • **Computer systems organization → Real-time systems**; **Dependable and fault-tolerant systems and networks;**

Additional Key Words and Phrases: Real-time fault tolerance, Apache ZooKeeper, response time analysis

ACM Reference format:
Haoran Li, Chenyang Lu, and Christopher D. Gill. 2021. RT-ZooKeeper: Taming the Recovery Latency of a Coordination Service. *ACM Trans. Embedd. Comput. Syst.* 20, 5s, Article 103 (September 2021), 22 pages.
https://doi.org/10.1145/3477034

1 INTRODUCTION

Distributed applications require different forms of coordination services (e.g., leader election, naming, global configuration, group membership, and synchronization). However, it is tedious and error-prone to design and implement such coordination features from scratch for a distributed

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2021.
This research was sponsored, in part, by NSF through grant 1646579 (CPS), and by the Fullgraf Foundation.
Authors' address: H. Li, C. Lu, and C. D. Gill, Washington University in St. Louis, Department of Computer Science and Engineering, 1 Brookings Drive, St. Louis, MO 63130-4899, USA.

1539-9087/2021/09-ART103 $15.00
https://doi.org/10.1145/3477034

application, and so reusable fault-tolerant coordination services, like ZooKeeper [12], have been widely used to develop distributed applications and services in cloud environments [30]: e.g., Kafka [20] leverages ZooKeeper for topic creation, electing a broker leader, mapping topic partition pairs, and monitoring topology changes; and Hadoop [33], Hive [35], Flink [6], and Storm [36] use ZooKeeper for group management, naming, and global configuration.

Modern embedded systems such as Internet of Things (IoT) are embracing edge clouds as local computing platforms to support sophisticated applications. In contrast to centralized clouds, edge clouds are targeted at time-sensitive applications given their physical proximity to IoT devices. Industrial edge cloud platforms often reuse mature middleware services well established in cloud computing. For example, Dell's Confluent Platform [8] supports edge analytics for industrial IoT through a suite of cloud services (e.g., Kafka) that depend on ZooKeeper. Similarly, Microsoft's Azure IoT Hubs employs Kafka for streaming telemetry data to downstream applications.

Many time-sensitive IoT applications have soft latency requirements. For example, an edge analytics application may expect sensor data to be delivered with desired latency to produce analytic results based on up-to-date states of the physical plant. The application can tolerate delayed messages by estimating sensor data based on recent readings. However, excessive delays (e.g., while the underlying services recovery from failures) may lead to degraded accuracy of the analytics due to outdated state information about the plant. For such applications, hard latency guarantees are often too costly to be practical in edge clouds due to their complex software stack, commercial-off-the-shelf hardware, and open operating environments. Instead, it is usually sufficient to meet desired latency based on empirical measurements.

We use Apache Zookeeper, the prevailing open-source coordination service, as a representative example through which to investigate how to achieve coordination with constrained recovery latency in edge computing environments. ZooKeeper comprises a group of replicated servers to provide fault-tolerant coordination services. It relies on the ZooKeeper Atomic Broadcast (ZAB) [16] protocol, a state of the art distributed protocol for synchronizing replicas against the leader, managing database update transactions, and recovering from a crashed state to a valid state. However, although ZooKeeper offers fault-tolerance, its service recovery time on a leader failure can be excessive for many time-sensitive applications, thus hindering their real-time performance.

In this paper, we analyze the ZAB protocol with a focus on its recovery procedures following leader failure, and identify limitations in leader election and recovery in terms of latency. To reduce latency in recovering from failures, we develop **RT-ZooKeeper (RTZK)**, a new coordination service featuring a set of novel protocols and mechanisms to drastically reduce the recover latency of ZooKeeper. Specifically, this paper makes the following main contributions.

- We identify the problems in the ZooKeeper design that cause excessive recovery delays after a leader failure;
- We design RT-ZooKeeper, which overcomes ZooKeeper's limitations and achieves fast failure recovery through three new features including (1) a *Fast-Convergence Election (FCE)* protocol, (2) a *Quorum-Channel Notification (QCN)* mechanism, and (3) a *Distributed Epoch Persistence (DEP)* protocol;
- We implement RT-ZooKeeper based on ZooKeeper version 3.5.8 and present empirical benchmarks showing RT-ZooKeeper shortens the maximum recovery latency by 91% in comparison to ZooKeeper;
- We present case studies involving Kafka, a widely used messaging service that relies on ZooKeeper, which demonstrate that RT-ZooKeeper can significantly shorten recovery latency, directly benefiting the messaging service provided by Kafka.

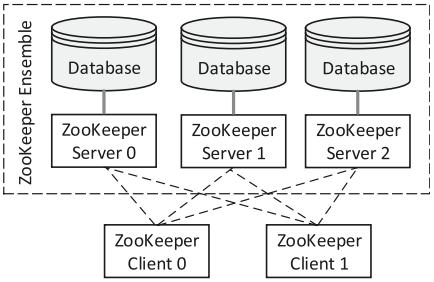


Fig. 1. ZooKeeper: Ensemble, Servers, and Clients.

2 OVERVIEW OF ZOOKEEPER

In this section, we provide an overview of ZooKeeper and its underlying ZooKeeper Atomic Broadcast (ZAB) protocol as background for this work.

2.1 ZooKeeper

Figure 1 illustrates the ZooKeeper service, which achieves fault tolerance by duplicating its in-memory database; each replica is handled by a *ZooKeeper server*. ZooKeeper maintains data synchronization among its replicas, and distributed applications use *ZooKeeper clients* to access the database.

Failure Model. Each ZooKeeper server follows the crash-recovery model [16]: a server can suffer from a *fail-stop failure* and crash at an arbitrary time, and then can recover after an additional interval. A ZooKeeper *ensemble* comprises N ZooKeeper servers $\Gamma = \{p_0, p_1, \ldots, p_{N-1}\}$. A *quorum* of Γ is a subset $Q \subseteq \Gamma$, where $|Q| > \frac{N}{2}$. Any two quorums have a non-empty intersection. The ZooKeeper service is available as long as a quorum (majority) of the ensemble is available. The number of servers in an ensemble is often an odd number, $N = 2f + 1$, which means f server crashes can be tolerated in this N-server ensemble. One of the servers works as the *leader* in the quorum, while the others are *followers*. Designed to handle server failures, ZooKeeper assumes any two servers can communicate with each other and the network is reliable [9, 16]. In practice, ZooKeeper usually relies on the underlying TCP protocol and redundant network topology to provide reliable communication.

Clients Feature Automatic Re-connection. By default, each client knows all the "entrances", i.e., the IP addresses and ports, of all ZooKeeper servers. Once a client connects to the ZooKeeper service, a ZooKeeper server will establish a *session* for the client and persist the session Id among all ZooKeeper servers. When a connection fails (due to a ZooKeeper server crash), the session will remain available for a while; the client will automatically try an alternative entrance for the ZooKeeper service before the session expires.

Write Requests are Logged as Transactions. A client can read from and write to the database by sending requests to a ZooKeeper server. Any ZooKeeper server, regardless its role (as a leader or a follower), can handle read requests locally. However, a write request must be forwarded to the leader. The leader is responsible for updating and synchronising the database across the ZooKeeper ensemble. Each change and update is logged as a *transaction* associated with a *transaction identifier*, z, which is a two-tuple (e, c): e is an *epoch* number, distinguishing the leader, as a ZooKeeper ensemble may change its leader during its lifetime; and c is a unique *sequence* number assigned to a particular committed transaction during epoch e. Whenever a new leader starts a new epoch, the sequence number c is reset to 0. Transactions can be ordered by z. Given two different transactions, $z = (e, c)$ and $z' = (e', c')$, we define $z \prec z'$, if $e < e'$, or $e = e' \wedge c < c'$.

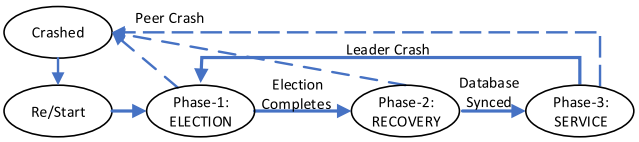


Fig. 2. ZAB: Phase Transition Diagram.

2.2 ZooKeeper Atomic Broadcast (ZAB) Protocol

ZAB is a crash-recovery atomic broadcast protocol [16] that is the most fundamental part of the ZooKeeper coordination service. ZAB helps the ensemble to maintain a mutually consistent state, especially when recovering from crashes. As none of the Apache ZooKeeper implementations strictly follows the ZAB protocol originally published in [28], in this paper we focus on the latest mainstream protocol ZAB-1.0 [29] and analyse it based on its implementation in ZooKeeper version 3.5.8.

Figure 2 shows a three-phase transition diagram for each ZooKeeper server. The ultimate goal of each ZooKeeper server is to reach Phase-3, SERVICE: only a ZooKeeper server in the SERVICE phase allows connections from clients and processes read and write requests.

Phase-1: ELECTION. When a server starts (or its leader is lost), it enters the ELECTION phase, running the leader election algorithm, and finally determining itself as either a follower or the new leader: the server with the latest (largest) transaction id, z, is elected as the new leader. The elected leader always starts a new *epoch*. The new epoch's value e, is always larger than any older epoch number. Leader election can converge as long as a quorum of peers is alive.

Phase-2: RECOVERY. After completing the ELECTION phase successfully, the leader executes a recovery protocol to ensure the followers' views of the database are synchronized with the leader's. When a peer's database has been synchronized in the RECOVERY phase, it enters the SERVICE phase and allows clients to connect.

A peer can crash at any arbitrary time in any phase. Usually, a peer's crash will not affect the availability of the entire service, except in two cases: either the leader crashes, or a follower crashes resulting in the number of live servers being less than $\frac{N}{2}$, such that no quorum can be achieved. In either case, all live peers transition to the ELECTION phase.

In this paper, we focus on the case where the leader fails and there remain at least $\frac{N}{2}$ servers alive that can form a quorum. In the case when a quorum becomes no longer available, the service recovery latency is usually dominated by the recovery of the crashed servers themselves.

3 LIMITATIONS OF ZOOKEEPER AND SOLUTIONS IN RT-ZOOKEEPER

In this section, we analyze ZooKeeper's policies and mechanisms for leader election and recovery, and identify their limitations that lead to prolonged service recovery after a leader failure. For each of the limitations identified, we propose a solution adopted in RT-ZooKeeper. While the original ZooKeeper may be deployed in a public cloud with potentially long communication latency, RT-ZooKeeper is tailored for an edge cloud that comprises a small number of servers connected by a LAN where communication latency is usually short. RT-ZooKeeper employs new protocols and mechanisms that exploit the characteristics of edge cloud platforms to reduce recovery latency.

3.1 Phase-1: Leader Election

3.1.1 Limitations of ZooKeeper. The ZAB protocol satisfies a property called *Primary Order* that ensures the correct ordering of state changes in systems where the leader may change (e.g. due to failure of the previous leader). The ZAB protocol achieves Primary Order as follows. The

new-elected leader must ensure it has the latest transaction among the living peers, before it tries to synchronize the followers with its data. The *Fast Leader Election (FLE)* algorithm ensures the elected leader will be the one that persists the most recent transaction. To implement FLE, ZooKeeper extends a well known distributed algorithm called OptFloodMax [25]. A distributed algorithm may be designed for a synchronous network or an asynchronous network. In a synchronous network, peers execute operations in each step simultaneously; in contrast, in asynchronous networks peers execute operations autonomously according to incoming events and their own states, as defined in [26, 27]. The original OptFloodMax algorithm was intended for synchronous networks [26]. To handle a realistic asynchronous network, ZooKeeper employs new mechanisms which however incur additional latency for convergence, as we show in this section.

In ZooKeeper, each server governs an atomic variable l, a `logicalclock`, to determine the election epoch. This variable is initialized by `currentEpoch` at a server's restart, and is increased when a peer goes into the `lookForLeader()` method for a new election.

Each server maintains a vote, $v = (z, i)$, a tuple consisting of the latest committed transaction id (z) of the server and the server's id (i). Given two different votes, $v = (z, i)$ and $v' = (z', i')$, we determine $v \prec v'$, if $z \prec z'$, or $z = z' \wedge i > i'$.

Importantly, for an extended OptFloodMax algorithm like FLE to converge on an asynchronous network, it needs a termination condition [27]. The FLE algorithm leverages a timeout mechanism for servers to reach the final decision on the elected leader. If a server does not receive any new vote for w ms, it will choose the server with the largest v (effectively the one with the largest z) as the leader. The election termination timeout value, w, is a fixed variable `finalizeWait` in the ZooKeeper implementation.

Specifically, each server executes the **FLE** protocol as follows:

Step 1. Broadcast Initial Vote. The server broadcasts its vote v, along with the election epoch l.

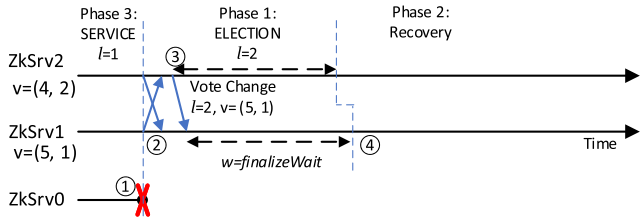
Step 2. Election Epoch Validation. Upon receiving a vote v', the server first checks if the election epoch of the received vote is right: $l' = l$. If $l' < l$, the vote will be ignored; if $l' > l$, the server will update its election epoch to l' and rebroadcast its vote.

Step 3. Flood the Maximum Vote. if $v \prec v'$, the server updates its vote: $v \leftarrow v'$, then re-broadcasts the updated vote, v'. If $v = v'$, the server tallies the vote. Otherwise, the server simply ignores the incoming vote.

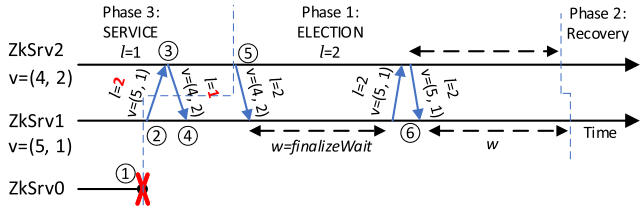
Step 4. Deduce the Leader. If the server does not receive any vote for $w =$ `finalizeWait` ms, the server tries to determine its role: If the tally of current vote does not reach a quorum, the server re-broadcasts the current vote v. If the tally of current votes $v = (z, i)$ has reached a quorum, the server determines server i to be the leader, while the others are followers.

If all the peers enter a phase-1 election simultaneously after the old leader's failure, the FLE algorithm only needs **one** `finalizeWait` round to converge. However, due to the asynchronous nature of real-world networks, each server may not enter phase-1 at the same time because of jitters in detecting the leader failure or in local processing. In such cases, we observe that FLE may result in significant latency penalties because servers had to wait for **multiple** rounds of `finalizeWait` before termination. We illustrate the delays induced by FLE under the two different scenarios in Figure 3(a)(b).

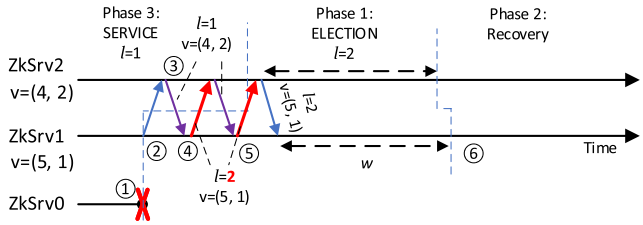
In the ideal case (Figure 3(a)), when the old leader fails, the followers transition from phase-3 (SERVICE) to phase-1 (ELECTION) simultaneously, starting a new election epoch with $l = 2$. As shown in Figure 3(a): ① The old leader dies. ② The two followers start broadcasting their initial votes. ③ ZkSrv2 receives a vote (5,1) which is greater than (4,2), changing its vote to (5,1) and broadcasting it again. Both ZkSrv1 and ZkSrv2 reach a quorum for (5,1). ④ After w ms, without receiving any new message, they both finish the election and transit to phase-2.



(a) Ideal case (FLE): Peers detect the leader failure simultaneously



(b) Asynchronous case (FLE): Peers detect the leader failure at different times



(c) Asynchronous case (FCE): re-send the notification

Fig. 3. Leader Elections in ZooKeeper (FLE) vs. RT-ZooKeeper (FCE).

However, the followers do not always detect the leader failure simultaneously. As a result, the peers may think they are in different election epochs. As shown in Figure 3(b): ① ZkSrv1 detects the leader failure earlier than ZkSrv2 and enters phase-1 with new election epoch $l = 2$. ② ZkSrv1 broadcasts its initial vote (5,1) to ZkSrv2, along with the $l = 2$. ③ However, since ZkSrv2 is still in phase-3, it has just sent a notification and claims it still follows the old leader ZkSrv0 by including $l = 1$. ④ However, ZkSrv1 ignores the notification since it has a higher `logicalclock` value. ⑤ After some time, ZkSrv2 detects the failure of the leader, transitions into phase-1, and broadcasts vote (4,2) with the increased `logicalclock` ($l = 2$), but the vote cannot override the initial vote (5,1) within ZkSrv1. None of the servers can reach a quorum based on their votes. ⑥ After w ms, ZkSrv1 decides to re-send its vote after ZkSrv2 changes its mind, waiting another w ms before phase-1 can be terminated. Thus, phase-1 experiences timeouts twice which enlarges the latency. Things can be even worse, if ZkSrv2 reaches a timeout faster than ZkSrv1 at stage 5 and re-send (4,2) again. In that scenario, ZkSrv1's timeout will be reset since it receives a new message, and wait an *additional* w ms.

3.1.2 RT-ZooKeeper: Fast Convergence Election. In order to avoid the delays caused by the multiple rounds of timeouts, we propose the *Fast Convergence Election (FCE)* protocol in RT-ZooKeeper. When a server receives a vote with a lower l `logicalclock`, it re-sends another notification

(including its current vote), as shown in Figure 3(c) ④. Such notifications will continue until the delayed server detects the old leader's failure (⑤). Thus, we can avoid additional w ms in Figure 3(b).

This mechanism to re-send election notifications is particularly suitable for an edge cloud platform. As the servers are connected by the same LAN, the jitter in their failure detection latencies is usually moderate. The proactive notification re-send mechanism allows the leader election process to converge quickly at the cost of a small number of re-sends. In our experiments, a single re-send usually allows the process to converge. In addition, we note that this notification re-send mechanism can be generally used to realize the popular OptFloodMax algorithm on an asynchronous network with fast convergence.

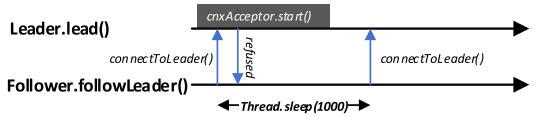
FCE builds on FLE to guarantee correctness. In FLE, the epoch value, which is implemented as the variable `logicalclock`, is used for recording how many times the leader has changed. When an old leader fails, its follower will come back to phase-1 and increase the epoch value, hence initiating a new round of leader election. Thus, the peers are always trying to elect a leader based on the highest epoch value. The correctness has been proved for the ZAB protocol (see Section IV in the ZAB paper [16]). The FCE protocol we proposed maintains building a new leader election on the highest epoch value. Thus it is still compliant with ZAB protocol's assumption and hence preserves the correctness guarantee.

3.2 Pre-Phase-2: Establish Quorum Channel

3.2.1 Limitations of ZooKeeper. Due to the different communication patterns and topology on different phases, ZooKeeper adopts two distinct communication channels: *election channel* for phase-1 and the *quorum channel* for the other phases. Server peers exchange votes in phase-1 via the election channel (bound to TCP port 3888 by default). During phase-1, each peer should broadcast its vote. The election channel establishes a complete graph: each node acts as a TCP server and accepts connections from other peers. The election server is always active once ZooKeeper has started. Other operations, including phase-2 recovery and phase-3 service, use the quorum channel (TCP port 2888 by default). After phase-1, the peers must have a leader. Each follower only exchanges messages and transactions with the leader, forming a star-topology: the leader accepts connections (as a server) and the followers connect to it (as clients). Note that the followers need to switch to the quorum channel after phase-1, since the quorum channel forms a star-topology: where the leader is in the center, exchanging message with each of the followers. In contrast, the leader election channel is a complete-graph, where there is no "center" or "leader".

At the end of phase-1, the leader calls the `lead()` method, then, calling `cnxAcceptor.start()` to bring up a new thread to accept incoming connection requests. Meanwhile, a follower immediately executes `Follower.followLeader()` and connects to the leader by calling the `connectToLeader()` method. As shown in Figure 4(a), since the new leader has more work to do, it is very likely that a follower's first connection request is refused. In the vanilla ZooKeeper implementation, the follower then must wait for a **fixed 1000ms** interval before the next retry, which significantly enlarges the overall recovery latency. While the delay may be mitigated by shortening the timeout threshold, the need to connect twice inevitably increases the time it takes to establish the quorum channel.

3.2.2 RT-ZooKeeper: Quorum Channel Notification. To reduce the latency for establishing quorum channel, RT-ZooKeeper avoids the need to reconnect to the quorum channel. Since the leader election channel (on TCP port 3888) is already established, RT-ZooKeeper exploits that channel for Quorum Channel Notification (QCN), letting the leader broadcast a *CNX_READY* packet to inform the follower once the new quorum channel is ready (Figure 4(b)), and thus avoid unnecessary waiting time and reconnection.



(a) Vanilla ZooKeeper: 1000ms re-connection retry

(b) Patch: Exploit the existing 3888 channel

Fig. 4. Follower: Connect to the Lead via TCP Port: 2888.

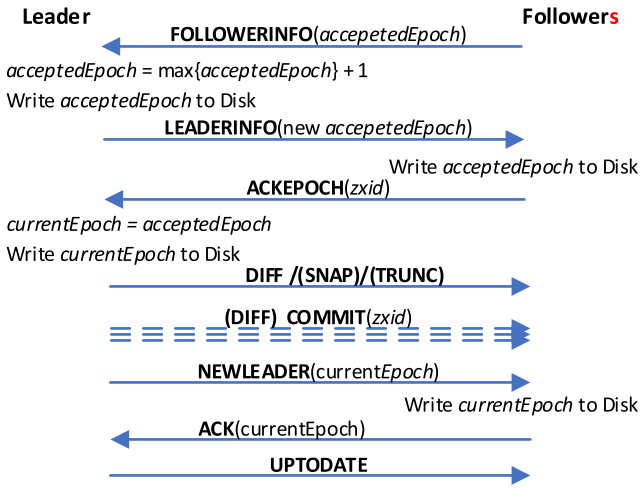


Fig. 5. Protocol of Phase-2 RECOVERY.

3.3 Phase-2: Recovery

3.3.1 Limitations of ZooKeeper. Figure 5 illustrates ZooKeeper's protocol in phase-2, RECOVERY. After electing the leader in phase-1, the quorum channel is established. Each follower then connects to the leader and sends a *FOLLOWERINFO* packet to the leader, reporting its `acceptedEpoch`. When the leader receives a quorum of `FOLLOWERINFO` packets (including the one the leader sent to itself), it generates a new `acceptedEpoch` whose value is greater than any from the receiving set. The leader then *persists* the new `acceptedEpoch` to disk. The leader replies to each follower with the new `acceptedEpoch` encapsulated within a *LEADERINFO* packet. Upon receiving the packet, the follower updates the `acceptedEpoch` and *persists* it to disk. The follower acknowledges the leader with *ACKEPOCH* and informs the leader of the latest `zxid` of the follower's database. The leader collects a quorum of acknowledgements, then updates its `currentEpoch` and *persists* it to disk. According to the leader election in phase-1, the leader should have the most recent `zxid`. As a result, the leader can initiate the database synchronization by sending a *DIFF* packet, followed by several *COMMIT* packets, until each follower is synchronized. Then, the leader sends a *NEWLEADER* packet to update the `currentEpoch` value, to the followers. Followers *persist* the `currentEpoch` on disk, sending an *ACK* back to the leader. Upon receiving a quorum of *ACK*s,

Table 1. EpochTable: Structure

Field	Type	Note
version	long	the version of this table
numServers	int	the size of the ensemble
acceptedEpochList	long[]	acceptedEpoch values of all servers
currentEpochList	long[]	currentEpoch values of all servers

the leader finally sends an *UPTODATE* packet and goes into phase-3, SERVICE. Each follower also transitions into the SERVICE phase after receiving the *UPTODATE* packet.

As shown in Figure 5, the phase-2 procedure involves four disk I/O operations for persisting `acceptedEpoch` and `currentEpoch` in both the leader and followers. Our empirical results show that significant latency is introduced by such disk I/O (see "RTZK-L Phase-2" in Figure 8(d)), especially because the two variables are stored in distinct files. Each file write operation can take as long as 300ms.

We note that the `acceptedEpoch` and `currentEpoch` variables play important roles in recovery: a server recovering from a crash must know the last epoch it accepted and was in, to deduce its status when it suffered a failure, and to take part successfully in leader election. It is therefore crucial to persist the variables.[1]

3.3.2 RT-ZooKeeper: Distributed Epoch Persistence. To avoid the excessive latency introduced by disk I/O, we propose the *Distributed Epoch Persistence (DEP)* protocol, persisting the epoch values (i.e. `acceptedEpoch` and `currentEpoch`) among a quorum of ZooKeeper servers. Our approach is based on the observation that, in an edge computing environment with a high-speed LAN, communication latency between local servers is usually smaller than that of disk I/O. To realize DEP, we conduct holistic modifications in both phase-1 and phase-2 of the vanilla ZAB protocol.

In DEP, each ZooKeeper server maintains an *EpochTable*, as shown in Table 1, which contains global information about the `acceptedEpoch` and *currentEpoch* of all the servers in the current ensemble. We change the means to access the two variables.

Phase-1 Modification. Each server needs to get the `currentEpoch`, to initialize l = `logicalclock` before staring leader election. In the ZAB protocol, each server accesses its disk to read the persisted value from a specified file. In the DEP protocol, before entering the election, each server runs an OptFloodMax algorithm (with the extensions introduced in FCE) to reach a consensus on the EpochTable. The servers choose the one with the latest (greatest) `version` as the consensus result. Then the servers can retrieve the epoch value from the already synchronized and up to date EpochTable.

Phase-2 Modification. In phase-2, ZooKeeper servers change and persist their epoch values. In ZAB, each ZooKeeper simply writes the new value to disk and sends an acknowledgement. When using DEP, we allow only the leader to modify the contents of its EpochTable, while the followers can only propose their variable changes to the leader. This can be done via a two-step commit protocol: (1) the leader accumulates the changes from all the followers (in a quorum). Then, the leader generates a new EpochTable with corresponding updates, including a new version number, broadcasting the EpochTable to the followers; and (2) upon receiving a new EpochTable, the follower commits it, sending an ACK to the leader. Once it has received a quorum of ACKs, the leader continues the ZAB protocol.

[1]The early implementation of ZAB, ZAB-Pre-1.0 [15], partially omitted these variables: it only recorded the epoch e in the latest persisted z, resulting in some "blocking level" bugs [14, 19] that were not completely resolved until ZAB-1.0 [29], which re-implements the handling of these two variables, in ZooKeeper version 3.4.

Table 2. Symbol Definition for the Timing Model

Symbol	Definition
T	Overall recovery latency for RTZK
T_e	Phase-1 latency for RTZK
T_o	latency for running OptFloodMax algorithm
T_r	Phase-2 latency for RTZK
w	`finalizeWait` value
j	jitter of leader failure detection latency
d	Upper-bound of message transmission latency
h	Message processing latency
h_r	Message processing latency for phase-2 of RTZK
b	Disk I/O latency
s_e, s_r	Initialization latency for phase-1/phase-2

Persisting Epoch Values. We use the same channel as leader election (TCP port 3888) for delivering our EpochTable-related messages, and implement the function handling them within the `FastLeaderElection` class, to address the following design considerations: 1. The latest EpochTable needs to be detected before leader election, so the quorum channel (TCP port 2888), which is established in pre-phase-2, cannot be used for this. 2. The latest EpochTable detection is based on the same algorithm as the FCE (OptFloodMax). 3. The application-level packet framing class, `ToSend`, has an enumerated type field `mType`, which facilitates message extension and multiplexing.

We add a `lookForLatestEpochTable()` method, for detecting and synchronizing the EpochTable before running leader election. It is similar in structure to the `lookForLeader()` method, since they both implement the OptFloodMax algorithm in a similar way. To support broadcasting the EpochTable via the election TCP channel, we modified `WorkSender.process()`, the message sending method, allowing it to frame two different message types: `notification` and `epochtable`, for FLE and EpochTable detection, respectively (based on `mType`, which is in the head of the frame). We also modified the message receiver and its forwarding thread (`WorkerReceiver.run()`): by parsing the message header, the receiver can forward the messages to the corresponding processing queue, for polling by `lookForLatestEpochTable()` or `lookForLeader()`.

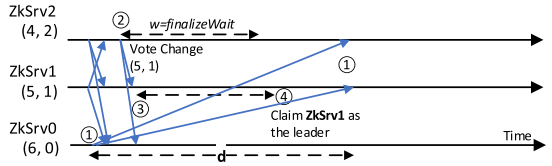
4 RECOVERY TIME ANALYSIS

In this section, we analyze the recovery latency of ZooKeeper and RT-ZooKeeper. As RT-ZooKeeper is targeted at time-sensitive applications in general instead of strictly hard real-time applications, the analysis is not intended to provide worst-case bounds or hard guarantees. Instead, it aims to establish approximate timing models and provide guidance for configuring the service.

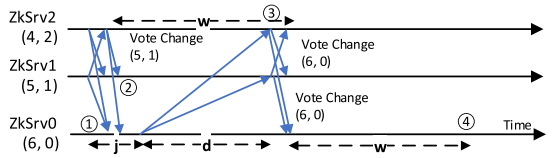
As shown in the phase transition diagram (Figure 2), the overall recovery latency T comprises two parts, the phase-1 election latency T_e and the phase-2 recovery latency T_r:

$$T = T_e + T_r. \tag{1}$$

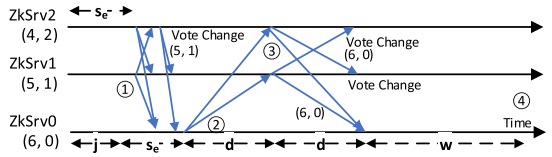
We start with the model of phase-1 election latency, revealing how jitter and communication delay can affect the model and then discussing the timing behaviour of OptFloodMax implemented for FCE. We then model the phase-2 and communication costs for maintaining the EpochTable via the DEP protocol.



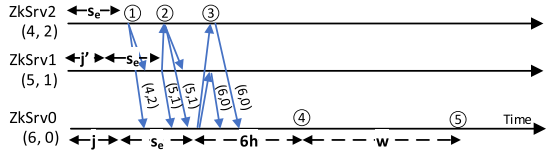
(a) A short `finalizeWait`, $w < d$, results in a false leader:



(b) A proper `finalizeWait`, $w > d + j$, which considers the jitter



(c) Cloud Environment: OptFloodMax $T_{o'}$ dominated by d



(d) LAN Environment: OptFloodMax T_o dominated by h

Fig. 6. Timing Model of Leader Election.

4.1 Timing Model for Phase-1

Duration of finalizeWait. In vanilla ZooKeeper, w has a hard-coded fixed value: 200ms. We argue that this value is unnecessarily large for a local area network (LAN) environment, where the transmission latency d can be significantly less than 200ms. However, w cannot be too short either, or the ensemble may elect the wrong leader, which would significantly increase phase-2 recovery latency. For example, in Figure 6(a), server ZkSrv0 has the most recent transaction and hence should be the new leader. However, ① ZkSrv0's messages suffer from a long transmission delay. ②③ ZkSrv1 and ZkSrv2 can reach a quorum (2 of 3) with the vote (5,1) being the maximum among them. ④ After w time units without receiving new votes, ZkSrv1 and ZkSrv2 will claim ZkSrv1 as the (false) leader.

It is not sufficient to just have $w > d$, since the ZooKeeper servers may not start leader election at exactly the same time. Instead, we can use a value j to represent the jitter in their collective start times. As shown in Figure 6(b): ① ZkSrv0 suffers from both jitter j and long transmission delay d. ② ZkSrv0 ignores the votes from other two servers. ③ We need a sufficiently large w to ensure ZkSrv1 and ZkSrv2 receive ZkSrv0's vote before they start deducing the leader. Hence, we require:

$$w > j + d, \tag{2}$$

to avoid the false leader scenario. ④ ZkSrv1 and ZkSrv2 change their votes to (6,0) and rebroadcast them and everyone reaches a quorum; the procedure converges.

Estimate OptFloodMax Latency. In a public cloud scenario (Figure 6(c)), where the transmission latency can be significantly larger than the message processing latency (e.g., where d can be more than 100ms and h can be several ms), the latency, T'_o, for running the OptFloodMax algorithm once can be modeled as:

$$T_{o'} = 2d + w. \tag{3}$$

① It takes a constant initialization time s_e before running the actual OptFloodMax algorithm. ② In addition, ZkSrv0 suffers from both the jitter and long transmission delay d. ③ Another transmission delay d is introduced when ZkSrv1 and ZkSrv2 wants to rebroadcast (6, 0). ④ The servers need to wait w for finally finishing the election.

However, in a LAN setting where d is less then 10ms, and usually is less than 1ms, the model in Equation (3) does not work since the single message processing time h can affect the overall leader election time significantly. Thus, we use a different model (T_o) for this scenario:

$$T_o = \frac{N(N+1)}{2}h + w. \tag{4}$$

The first term indicates the potential message processing time of the leader. In an ensemble of n servers, the server with n^{th} largest vote can change its mind $n-1$ times, thus broadcasting at most n messages to the network. As a result, the leader will receive $1 + 2 + \cdots + N$ messages in leader election process. Figure 6(d) shows an example: ① ZkSrv1 broadcast its initial vote (4,2) first, but was ignored by the other two. ② After a jitter interval j', ZkSrv2 broadcasts its vote (5,1), resulting in a vote change in ZkSrv1. ③ ZkSrv0 broadcasts its vote and causes the other two to change their votes. ④ ZkSrv0 receives $3 \times (3+1)/2 = 6$ messages from other peers, 5 from other peers plus 1 when ZkSrv0 broadcasts (6,0) to itself. ⑤ The procedure converges.

RT-ZooKeeper needs to run the OptFloodMax algorithm twice in phase-1, once for EpochTable Detection, and once for leader election, but we only need to count initialization (including jitter) once:

$$T_e = s_e + j + 2T_o. \tag{5}$$

4.2 Timing Model for Phase-2

As shown in Figure 5, the ensemble first reaches a consensus on a new epoch, which is greater than any previous epoch and must persist. Though the recovery phase involves several rounds of message exchanges, the most time consuming part is to persist the `acceptedEpoch` and `currentEpoch` values to disk: The leader and each of the followers have to write to the disk two times, resulting in 4x disk I/O latency.

In phase-3, ZooKeeper keeps the transaction synchronous among all peers. As a result, when a leader failure occurs, the unsynchronized transactions can be bounded. Thus, the timing model for phase-2 of vanilla ZooKeeper can be relatively simple:

$$T_{r'} = s_r + 4b. \tag{6}$$

Note that b stands for the I/O latency when writing either `acceptedEpoch` or `currentEpoch` to disk, while initialization and other fixed message communication and processing latency can be combined into a single constant s_r.

The DEP protocol leverages a two-phase-commit to update the *EpochTable*. Note that we remove the latency term $4b$ from Equation (6) by avoiding disk I/O. The relationship between the size of

the ensemble and additional epoch messages to be processed is linear. Thus, the phase-2 timing model for RT-ZooKeeper is:

$$T_r = s_r + Nh_r. \tag{7}$$

From Equation (8), (5), (4), and (7), we have:

$$T = s_e + j + 2\left(\frac{N(N+1)}{2}h + w\right) + s_r + Nh_r. \tag{8}$$

5 LIMITATIONS

RT-ZooKeeper is not designed to provide hard latency guarantees. The nature of ZooKeeper – running on JVM, complicated multi-thread synchronization, and standard TCP/IP protocol stack and network technologies – makes it far from being a hard real-time coordination solution. RT-ZooKeeper, while significantly reducing failure recovery latency, remains a service suitable for applications with soft latency requirements that can be satisfied through empirical measurements. Consequently, the timing model we established is intended as an estimation of recovery latency rather than a hard bound for timeliness guarantees. The timing model can provide guidance for configuring RT-ZooKeeper to help developers achieve desired recovery latency empirically.

RT-ZooKeeper focuses on reducing failure recovery latency. It is not designed to improve latency during normal, fail-free operations. As an in-memory datastore, ZooKeeper has a millisecond-level latency when dealing with regular transactional operations during phase-3. In contrast, the recovery latency observed in our evaluation was second-level. Hence, recovery latency is significantly larger than the latency during regular operations. Therefore, this paper focuses on shortening the recovery latency, which dominates the maximum latency experienced by the application.

6 EMPIRICAL EVALUATION

In this section, we use a micro-benchmark on a real testbed to measure and evaluate the recovery latency for three ZooKeeper variants:

- **Vanilla ZooKeeper.** The original ZooKeeper 3.5.8 without any patch (whose `finalizeWait`, $w = 200$ms).
- **RT-ZooKeeper Lite (RTZK-L).** ZooKeeper with partial features of RT-ZooKeeper: only FCE and QCN are enabled.
- **RT-ZooKeeper (RTZK).** The full-fledged RT-ZooKeeper with all three features (FCE, QCN, and DEP) enabled.

RTZK-L is configured for evaluation purposes: to differentiate the performance gains from different features. We only enable FCE and QCN in RTZK-L since those two protocols and modifications are well-isolated and independent: their performance gains can be easily differentiated by comparing to the phase-1 and phase-2 latency from vanilla ZooKeeper, respectively. Using RTZK-L as a baseline, we can then evaluate pros and cons for DEP, which involves modifications in both phases. We also evaluate whether the empirical maximum latency results for RTZK are bounded by the estimations from Equation (8).

We conducted experiments on a testbed with three physical machines working as edge hosts. Each host has one Intel E5-2680v4 8-core CPU, 64 GB memory, and 1TB 7200 RPM-HDD. We disabled hyper-threading and power saving features and fixed the CPU frequency at 2.1 GHz to improve predictability, as in [18, 39, 40]. As virtualization is often used in edge computing environments to facilitate deployment and other practical considerations, each physical host runs a Xen 4.12.0 hypervisor to consolidate multiple virtual machines (VMs). As shown in Figure 7, each ZooKeeper server (or Kafka broker) is encapsulated with a Linux VM (kernel version 5.4.0) and

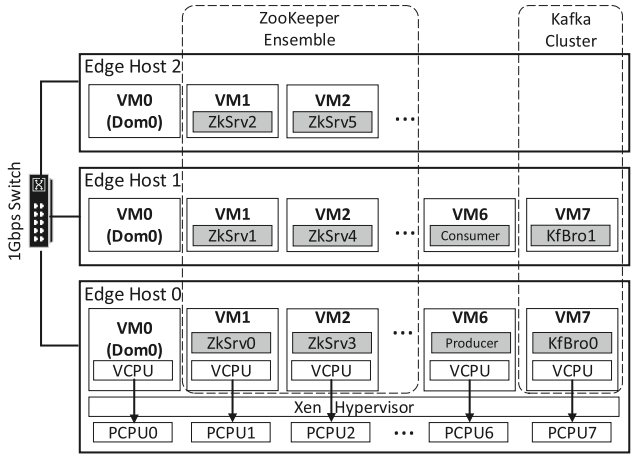


Fig. 7. Testbed for ZooKeeper Recovery Evaluation.

Table 3. Parameters for Latency Estimation

Parameter	s_e	j	w	h	s_r	h_r
Value(ms)	35	13	20	3	55	25

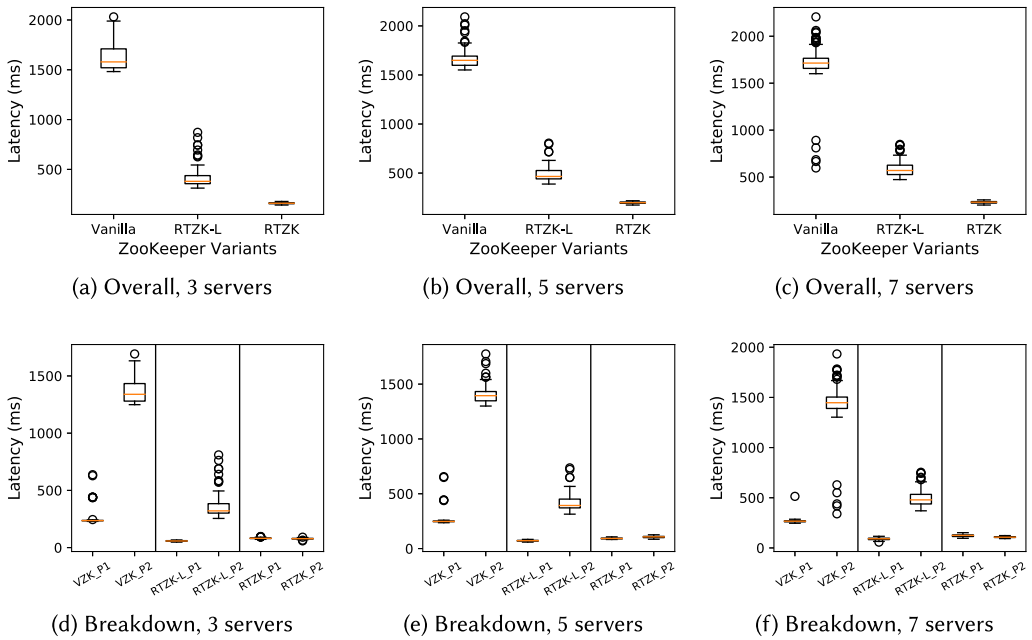
consolidated on one of the physical hosts. Each VM has one VCPU which is pinned on a dedicated PCPU. The administrative VM of Xen, Dom 0, has one VCPU which is pinned onto PCPU0. The physical hosts are connected to a 1Gbps switch, and all the VMs are in the same LAN. All ZooKeeper variants we evaluated are based on version 3.5.8.

As shown in Figure 7, each ZooKeeper ensemble in our evaluations consists of up to seven ZooKeeper servers, whose ids range from 0 to 6 (*ZkSrv0 to ZkSrv6*), which we allocate to physical hosts in a round-robin fashion. We bring up a ZooKeeper ensemble, kill the leader to trigger the ZooKeeper recovery procedure, and then measure the recovery latency: for each of the three ZooKeeper variants, we repeat that measurement procedure 100 times.

We measured the maximum transmission latency at $d = 3$ms, and the maximum jitter at $j = 13$ms for failure detection. Thus we choose the `finalizeWait`, $w = 20$ms following Equation (2) and using the other parameters shown in Table 3.

We ran the same experiments using ensembles consisting of 3, 5, or 7 ZooKeeper servers: although it is possible to have more servers, running a ZooKeeper ensemble with more than seven servers is rare in production settings, even in a public cloud environment. For example, CloudKarafka allows up to seven ZooKeeper servers [37] and Solr recommends no more than five ZooKeeper servers [24]. Moreover, a seven-server-ensemble can easily serve more than $1,000$ nodes which is beyond the scale of a normal edge cloud: we argue that especially since the number of compute nodes in an edge cloud is much less than in a public production environment, it is reasonable to test against those commonplace settings.

Figures 8(a) to 8(c) show the overall latency distributions for the three ZooKeeper variants (vanilla ZooKeeper, RTZK-L, and RTZK) with a 3, 5, or 7 server ensemble. We observe that RTZK-L outperforms vanilla ZooKeeper and that among the three variants RTZK has the best performance, significantly reducing maximum latency from 2031ms to less than 178ms, i.e., RTZK shortens latency by 92%.



(a) Overall, 3 servers (b) Overall, 5 servers (c) Overall, 7 servers

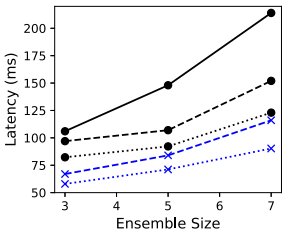
(d) Breakdown, 3 servers (e) Breakdown, 5 servers (f) Breakdown, 7 servers

Fig. 8. ZooKeeper Recovery Latency Distribution.

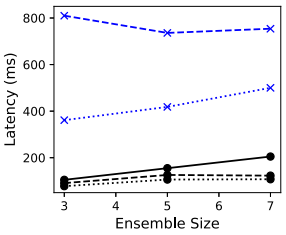
We further analyse the results by breaking down the overall latency for each ZooKeeper variant into two parts: phase-1 leader election latency and phase-2 database recovery latency, as shown in Figures 8(d) to 8(f). We make three observations based on those results: 1. The phase-1 leader election latency samples for vanilla ZooKeeper (*VZK_P1*) are clustered into three levels: 250ms, 450ms, and 650ms. This phenomenon is due to the unnecessary w = `finalizeWait` rounds, which we explained in Section 3.1. Our corresponding FCE protocol fixes this problem and reduces the w value, as we can see in *RTZK-L_P1*. 2. The phase-2 database recovery latency dominates the overall latency for vanilla ZooKeeper. RTZK-L improves phase-2 recovery latency (RTZK-L_P2) by avoiding the unnecessary one-second re-connection wait via the QCN mechanism, as we discussed in Section 3.2. However, even that optimized latency is still relatively long and has significant outliers due to disk operations. 3. RTZK avoids those disk operations by persisting those values among the quorum via the DEP protocol. RTZK thus significantly shortens the phase-2 recovery latency and achieves the best overall results among the three variants.

Given the parameters in Table 3, we calculate an estimated maximum overall (T), phase-1 (T_e), and phase-2 (T_r) latency, for RTZK based on Equation (8), (5), and (7), respectively. Note that since a server has crashed, the number of servers taking part in recovery is one less than the ensemble size. We show the estimated maximum latency (T_e, T_r, and T) in Figures 9(a) to 9(c), which provide estimated bounds for gauging RTZK's empirical results.

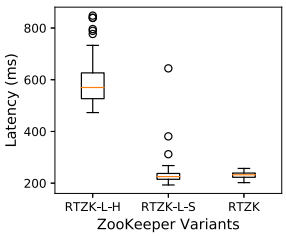
We note that RTZK achieves the best performance by avoiding disk operation latency (Figure 9(c)) but at a cost of more network operations. Hence the phase-1 leader election latency of RTZK is larger than the one for RTZK-L (Figure 9(a)). Specifically, RTZK-L only needs to run the *OptFloodMax* algorithm once for electing a leader, but RTZK needs to run it twice, once for synchronizing the `EpochTable`, and again for leader election. However, it is not a major concern in light of real-world ZooKeeper ensemble sizes, which are rarely greater than seven even in production [24, 37].



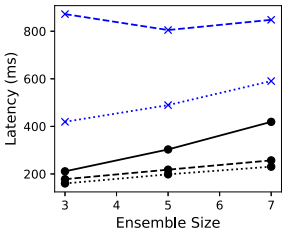
(a) Phase-1: Election



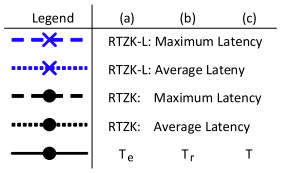
(b) Phase-2: Recovery



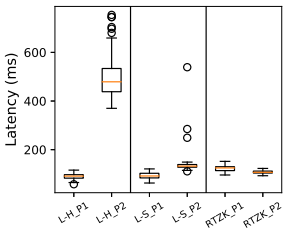
(a) Overall, 7 servers



(c) Overall



(d) Legends for (a),(b),(c)



(b) Breakdown, 7 servers

Fig. 9. Latency on different Variant/Ensemble.

Fig. 10. RTZK-Lite on SSD.

RTZK-Lite on SSD. To further evaluate the DEP protocol used by RT-ZooKeeper, we replaced the HDD disk for RTZK-Lite with an SSD disk.

Figure 10 shows the latency distribution of three different configurations on a seven-server-ensemble: RTZK-Lite on HDD (RTZK-L-H), RTZK-Lite on SSD (RTZK-L-S), and RTZK (which does not need disk I/O for accessing epoch values). RTZK-L-S shortens phase-2 latency in most cases (L-S_P2 in Figure 10(b)), compared to its HDD variant. However, it still suffers from excessive latency occasionally. Hence the maximum latency performance of RTZK-Lite, even when equipped with an SSD, may be too large. RTZK, in contrast yields much more consistent results, without outliers that significantly deviate from the majority of latency values.

7 CASE STUDY WITH KAFKA

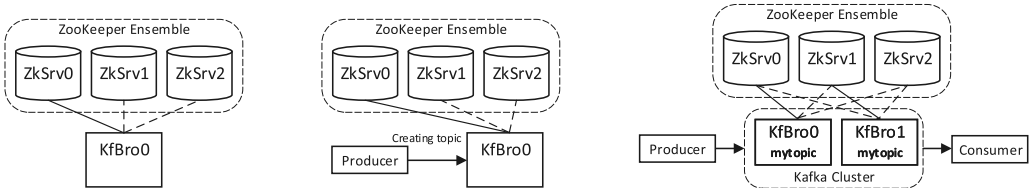
In this section, we present case studies involving a distributed real-time messaging service, Kafka, on the same testbed shown in Figure 7, with Kafka version 2.6.0. The Kafka brokers are ZooKeeper clients, using ZooKeeper for topic creation, leader election for brokers and topic partition pairs, and monitoring topology changes for the Kafka cluster.

We built a Kafka cluster with two Kafka Brokers (*KfBro0 and KfBro1*) distributed on two physical hosts. We also use other VMs for a Kafka message producer and consumer. We conducted three case studies of increasing complexity, to evaluate the benefits of low recovery latency using RT-ZooKeeper: 1. a Kafka re-connection latency test, 2. a Kafka topic creation latency test, and 3. a Kafka message end-to-end latency test. In each case study, we measure the corresponding operation's latency following ZooKeeper leader failure.

7.1 Kafka Re-connection

As shown in Figure 11(a), we use a ZooKeeper ensemble consisting of three servers, and a Kafka broker, which knows all the entry points of the ZooKeeper ensemble.

We conduct an experiment to measure the Kafka re-connection latency following ZooKeeper leader failure. In each run, we first kill the ZooKeeper leader. The ZooKeeper service then becomes



(a) Kafka Re-connection (b) Kafka Topic Creation (c) Kafka Message End-to-End Latency

Fig. 11. System Architecture for each Case Study.

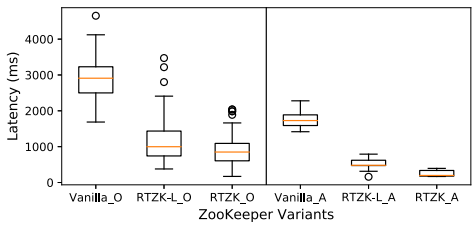


Fig. 12. Kafka Re-connection Overall Latency.

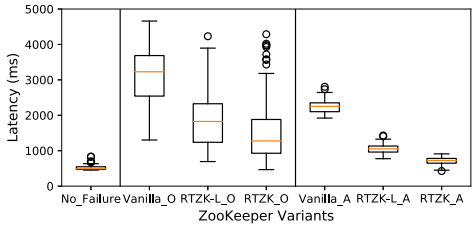


Fig. 13. Kafka Topic Creation Overall Latency.

temporarily unavailable and the Kafka broker loses its connection to the ensemble. After some time, the ZooKeeper service recovers. Meanwhile, the Kafka broker, as a ZooKeeper client, will keep trying to reconnect. The re-connection request cannot succeed until the ZooKeeper ensemble completes its recovery. We then measure the elapsed time from Kafka's lost connection, to its re-connection.

ZooKeeper Client Re-connection Strategy. First, we highlight that the client re-connection strategy is a significant factor for re-connection latency, due to two different innate strategies in the vanilla ZooKeeper client library. First, after failure of the client's connection to a ZooKeeper server, the client will wait for a randomized interval, ranging from 0 to 1000ms, before retrying. Second, vanilla ZooKeeper has a "delayed server redirection" mechanism: if a client tries to reconnect to a recovered ZooKeeper ensemble, but picks up the failed server to try to connect with, the client will incur an additional one second delay before it can try another server.

Aggressive Re-connection Strategy. Unfortunately, these re-connection delays introduced by the vanilla ZooKeeper client library can even eclipse the benefits of RTZK. Therefore, we patched the ZooKeeper client library to use significantly more responsive settings: the client will retry every 50ms, and will immediately try another entry point if it accidentally picks up a failed server.

We ran our experiments 100 times for each of six different ZooKeeper configurations: vanilla ZooKeeper, RTZK-L, and RTZK, each with either the original ZooKeeper client library or with the one patched to use our aggressive re-connection strategy.

Figure 12 shows the Kafka re-connection latency distributions for the different ZooKeeper configurations. We make four observations: 1. RTZK always outperforms the corresponding vanilla ZooKeeper and RTZK-L configurations in terms of both median and maximum latency, regardless of the re-connection strategy. 2. When using the original conservative re-connection strategy (labeled *_O), the Kafka broker is likely to take more time to reestablish the connection, hindering the potential benefit from using optimized versions of ZooKeeper (RTZK-L or RTZK). 3. The aggressive re-connection strategy (labeled as *_A) can efficiently alleviate the latency penalty introduced by the client, hence making the overall results close to what we report in Figure 8. Moreover, by combining the aggressive connection strategy with RT-ZooKeeper, we achieve the best performance

and significantly reduce latency. 4. Similarly low latency is not achieved by only applying an aggressive re-connection strategy with vanilla ZooKeeper, nor by only using RT-ZooKeeper without adopting the aggressive re-connection strategy.

7.2 Kafka Topic Creation

In this case study, we add a Kafka producer to create a new topic (Figure 11(b)). The topic's metadata is stored in ZooKeeper. Thus, ZooKeeper is on the critical path for topic creation.

We measure the topic creation latency under the following conditions: we first test the latency distribution in normal conditions, where no failure occurs; then, we intentionally kill the ZooKeeper leader during topic creation, to evaluate the impact of ZooKeeper recovery. We repeat the experiments for the three ZooKeeper variants: vanilla Zookeeper, RTZK-L, and RTZK, using the aggressive re-connection strategy.

Based on the results in Figure 13, we make two observations: 1. ZooKeeper failure does have negative impact on the Kafka topic creation latency: none of the configurations (with a ZooKeeper failure) achieves the optimal topic creation latency distribution seen without a ZooKeeper failure. 2. By combining the aggressive connection strategy with RTZK, we can alleviate such a negative impact to the greatest extent, significantly reducing latency when compared to other configurations.

7.3 Kafka Message End-to-End Latency

In this case study, we explore how ZooKeeper leader and Kafka leader failure affect message end-to-end latency.

As shown in Figure 11(c), we create a three-server ZooKeeper ensemble for a Kafka cluster consisting of two Kafka brokers. We create a topic "*mytopic*" with a replication factor of 2, i.e., Kafka is responsible to duplicate the message log on both Kafka brokers. After initialization, Kafka broker 0 and ZooKeeper server 0 are the leaders for the Kafka cluster and the ZooKeeper ensemble, respectively. A producer publishes a message to "mytopic" every 100ms (periodically), and a consumer subscribes to the same topic. We measure the end-to-end message latency under different circumstances.

Only The ZooKeeper Leader Fails. Kafka leverages the ZooKeeper service for topic creation, electing a new leader whenever a previous leader fails. However, for regular message routing, Kafka does not need to consult ZooKeeper for delivering messages. Hence, a failure of the ZooKeeper service will **not** affect the message end-to-end latency.

Only The Kafka Leader Fails. If the Kafka topic leader fails, the Kafka broker cannot deliver messages until the Kafka replica recognizes the failure of the old leader and establishes a new one. Kafka failure detection is handled by ZooKeeper: ZooKeeper establishes an entity called a *Session* to represent the existence of a living Kafka broker. The Kafka broker sends heartbeats to the ZooKeeper, indicating its existence. If the session times out, ZooKeeper treats the broker as failed and then notifies the other corresponding brokers.

The `tickTime` setting affects latency. Clearly, `zookeeper.session.timeout.ms`, the session timeout value in the Kafka broker configuration can significantly affect the failure detection latency: the lower that value, the faster ZooKeeper can detect the Kafka broker failure. The minimal session timeout value is gauged by the ZooKeeper `tickTime`, and can be at least twice that value. In the following experiment, we always set `zookeeper.session.timeout.ms` to be two times the value of the ZooKeeper `tickTime`.

We first ran the experiment with **only** a Kafka leader failure, but **without** a ZooKeeper leader failure. We measured the worst message end-to-end latency in each run. We performed 100 runs for each of the following ZooKeeper `tickTime` settings: {300, 500, 1000, or 2000} milliseconds. As

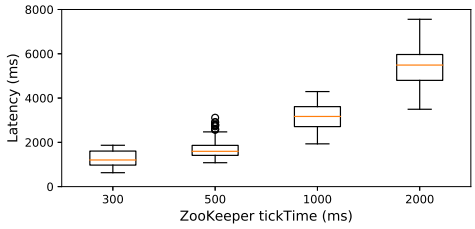


Fig. 14. E2E Latency: Only Kafka Leader Fails.

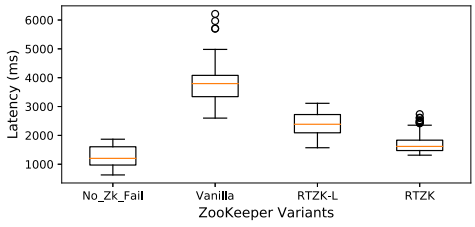


Fig. 15. E2E Latency: Both Kafka and ZK Leader Fail.

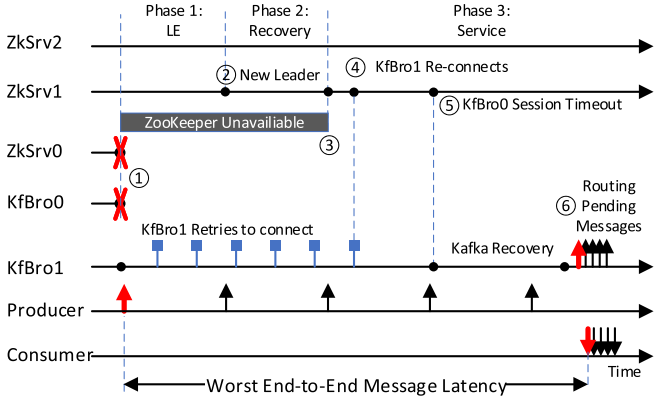


Fig. 16. Timeline for Measured Kafka End-to-End Message Latency with Kafka and ZooKeeper Leader Failures.

shown in Figure 14, even without ZooKeeper undergoing a recovery, the `tickTime` affects the latency significantly: the larger the `tickTime`, the worse the end-to-end latency.

Both the Kafka Leader and the ZooKeeper Leader Fail. In this case, ZooKeeper is on the critical path of the service: the other Kafka broker relies on ZooKeeper for the failure notification. Figure 16 shows the timeline for such a case: ① The Kafka leader (KfBro0) and the ZooKeeper leader (ZkSrv0) fail simultaneously. The other two ZooKeeper servers lose their leader, closing all connections of the client (KfBro1) and starting leader election. The other Kafka broker (KfBro1) loses its connection to ZooKeeper, and immediately tries to re-connect to the ZooKeeper ensemble (we assume the client adopts the "aggressive re-connection" strategy). ② The remaining ZooKeeper establishes a new ensemble consisting of two servers, elects a new leader, and starts phase-2 recovery. ③ The ZooKeeper ensemble recovery completes and resets the session timeout deadline. ④ KfBro1 finally reconnects to the recovered ZooKeeper service, avoiding a session timeout. ⑤ The KfBro0 session in the ZooKeeper ensemble must expire later, since KfBro0 had been dead. As a result, the ZooKeeper ensemble can send the notification for KfBro1. ⑥ KfBro1 finishes its recovery, including Kafka cluster leader election and data recovery, and can then process the pending messages.

The producer, regardless of the temporarily unavailable Kafka service, sends messages periodically. These messages cannot be routed until the Kafka cluster recovers. As a result, the messages sent just after the failure occurring suffer the longest latency. We record that for each run.

We set the ZooKeeper `tickTime` to 300ms, and the Kafka session timeout to 600ms. We made 100 runs for each ZooKeeper variant and show the distribution of the worst end-to-end latency in Figure 15. We make the following observations: 1. The only-Kafka-leader-failure case represents

the optimal lower bound for this experiment. The experiment results involving both Kafka and ZooKeeper failures can never outperform the one without ZooKeeper failure. 2. RT-ZooKeeper still achieves the best performance, by combining rapid recovery with the "aggressive re-connection" strategy.

Based on the micro benchmark and the three different real-world case studies presented in this section, we can draw the following conclusions: 1. FCE and QCN effectively shorten ZooKeeper recovery latency. 2. DEP further improves recovery latency by avoiding disk I/O. 3. The combination of RT-ZooKeeper with an aggressive client reconnection strategy is effective in constraining latency for services such as Kafka, which depend on ZooKeeper.

8 RELATED WORK

The research community has addressed fault tolerance of distributed systems in different ways. Protocols like Paxos [21] are widely used for building replicated state machines (RSMs) [22]: e.g., Gaios [4], S-Paxos [3], and Chubby [5]. ZooKeeper [12] and its ZAB [16] protocol address consistency issues for Paxos-based protocols: the primary fault can result in conditions that violate causal ordering, e.g., per the example found in the introduction of the ZAB paper [16]. The ZAB protocol resolves this issue by fulfilling the *Primary Order* property, which is necessary for database consistency while allowing primary failure.

Fault tolerance has also been investigated extensively for real-time systems. Earlier work addressed real-time performance in the presence of crash failures [1, 2]. DeCoRAM [1] provides a task allocation algorithm for replicated systems to meet real-time and fault-tolerance requirements. FLARe [2] maintains real-time performance through adaptive failover of real-time tasks to replicated servers. Fault-tolerance approaches for real-time systems involving transient and Byzantine faults have also been widely studied and developed [7, 10, 11, 23, 31, 32, 34]. These approaches are targeted at traditional real-time systems instead of distributed coordination services in edge clouds. Furthermore, none of the existing fault-tolerant approaches for real-time systems dealt with recovery latency associated with leader election.

With the emergence of edge computing, there has been recent effort to develop time-sensitive and fault-tolerant services in the edge cloud environment. FRAME [38] is a fault-tolerant real-time messaging service that exploits the tradeoff between message loss and soft latency requirements in an edge cloud environment. FRAME is a messaging service based on a standard primary-backup architecture. It does not address leader election and the associated recovery latency that can dominate the maximum latency experienced by edge applications.

ZooKeeper has received significant attention in the research community as a representative distributed coordination service. However, most existing research on ZooKeeper have focused on throughput and latency performance during normal operations (phase-3) [9, 13, 17], leaving recovery latency unaddressed. In contrast, RT-Zookeeper focuses on reducing the recovery latency of ZooKeeper for time-sensitive edge applications, with novel contributions addressing phase-1 and phase-2 that can greatly reduce recovery latency.

9 CONCLUSIONS

We have studied the recovery latency of fault-tolerant coordination services in edge computing environments, with a specific focus on ZooKeeper, the prevailing reliable coordination service. We focus on improving recovery latency following leader failure for ZooKeeper and identify bottlenecks that impede recovery. We then propose new protocols to alleviate those limitations. We implement those advances in RT-ZooKeeper based on ZooKeeper 3.5.8. Results of our empirical evaluations, including case studies using Kafka, demonstrate that our analyses and the improvements that arose from them can significantly reduce recovery latency. RT-ZooKeeper provides

appropriate support for latency-sensitive distributed applications running in edge computing environments. Targeting ZooKeeper as a representative system, this work highlights the importance of revisiting the fundamental design of cloud services from a latency perspective to support time-sensitive applications on edge computing platforms.

REFERENCES

[1] Jaiganesh Balasubramanian, Aniruddha Gokhale, Abhishek Dubey, Friedhelm Wolf, Chenyang Lu, Chris Gill, and Douglas Schmidt. 2010. Middleware for resource-aware deployment and configuration of fault-tolerant real-time systems. In *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, Stockholm, Sweden, 69–78.

[2] Jaiganesh Balasubramanian, Sumant Tambe, Chenyang Lu, and Aniruddha Gokhale. 2009. Adaptive failover for real-time middleware with passive replication. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09)*. IEEE, San Francisco, 118–127.

[3] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. 2012. S-paxos: Offloading the leader for high throughput state machine replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*. IEEE, Irvine, 111–120.

[4] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. 2011. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. NSDI'11, USENIX Conference on Networked Systems Design and Implementation*. ACM, Boston, 141–154.

[5] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*. ACM, Seattle, 335–350.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015), 1–1.

[7] Minyu Cui, Angeliki Kritikakou, Lei Mo, and Emmanuel Casseau. 2021. Fault-tolerant mapping of real-time parallel applications under multiple DVFS schemes. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'21)*. IEEE, Nashville, 387–399.

[8] Dell. Inc. 2020. Edge Analytics for Industry 4.0 with Confluent Platform. https://www.delltechnologies.com/asset/en-us/products/ready-solutions/technical-support/h18352-da-edge-iiot-confluent-dellencinfra-ra.pdf. Accessed: 2021-06-04.

[9] Ibrahim EL-Sanosi and Paul Ezhilchelvan. 2018. Improving zookeeper atomic broadcast performance when a server quorum never crashes. *EAI Endorsed Transactions on Energy Web* 5, 17 (2018), 1–1.

[10] Neeraj Gandhi, Edo Roth, Robert Gifford, Linh Thi Xuan Phan, and Andreas Haeberlen. 2020. Bounded-time recovery for distributed real-time systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, Sydney, 110–123.

[11] Arpan Gujarati, Sergey Bozhko, and Björn B. Brandenburg. 2020. Real-time replica consistency over ethernet with reliability bounds. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, Sydney, 376–389.

[12] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, Vol. 8. ACM, Boston, 1–1.

[13] EL-Sanosi Ibrahim and Paul Ezhilchelvan. 2017. Improving zookeeper atomic broadcast performance by coin tossing. In *European Workshop on Performance Engineering*. Springer, Berlin, 249–265.

[14] Flavio Junqueira. 2010. Last processed zxid set prematurely while establishing leadership. https://issues.apache.org/jira/browse/ZOOKEEPER-790. Accessed: 2020-10-01.

[15] Flavio Junqueira. 2013. Zab Pre 1.0. https://cwiki.apache.org/confluence/display/ZOOKEEPER/ Zab+Pre+1.0. Accessed: 2020-10-01.

[16] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, Hong Kong, 245–256.

[17] Babak Kalantari and André Schiper. 2013. Addressing the ZooKeeper synchronization inefficiency. In *International Conference on Distributed Computing and Networking*. Springer, Mumbai, 434–438.

[18] Hyoseung Kim and Ragunathan Rajkumar. 2016. Real-time cache management for multi-core virtualization. In *2016 International Conference on Embedded Software (EMSOFT)*. IEEE, Pittsburgh, 1–10.

[19] Mahadev Konar. 2011. Zookeeper servers should commit the new leader txn to their logs. https://issues.apache.org/jira/browse/ZOOKEEPER-335. Accessed: 2020-10-01.

[20] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. IEEE, Athens, 1–7.

[21] Leslie Lamport. 1998. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)* 16, 2 (1998), 133–169.
[22] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.
[23] Andrew Loveless, Ronald Dreslinski, Baris Kasikci, and Linh Thi Xuan Phan. 2021. IGOR: Accelerating byzantine fault tolerance for real-time systems with eager execution. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'21)*. IEEE, Nashville, 360–373.
[24] Lucidworks. 2018. Setting Up an External ZooKeeper Ensemble. https://doc.lucidworks.com/fusion-server/4.2/reference/solr-reference-guide/7.5.0/setting-up-an-external-zookeeper-ensemble.html. Accessed: 2020-10-01.
[25] Nancy A. Lynch. 1996. *Distributed algorithms*. Elsevier, San Francisco.
[26] Nancy A. Lynch. 1996. *Distributed algorithms*. Elsevier, San Francisco, Chapter Algorithms in General Synchronous Networks, 51–80.
[27] Nancy A. Lynch. 1996. *Distributed algorithms*. Elsevier, San Francisco, Chapter Leader Election in an Arbitrary Network, 495–496.
[28] André Medeiros. 2012. *ZooKeeper's atomic broadcast protocol: Theory and practice*. Technical Report. Technical report.
[29] Benjamin Reed. 2016. Zab 1.0. https://cwiki.apache.org/confluence/display/ZOOKEEPER/ Zab1.0. Accessed: 2020-10-01.
[30] Benjamin Reed and Norbert Kalmar. 2019. Applications Powered by ZooKeeper. https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy.
[31] Edo Roth and Andreas Haeberlen. 2021. Do not overpay for fault tolerance! In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'21)*. IEEE, Nashville, 151–160.
[32] Maurice Sebastian, Philip Axer, and Rolf Ernst. 2011. Utilizing hidden markov models for formal reliability analysis of real-time communication systems with errors. In *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*. IEEE, Pasadena, 79–88.
[33] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. IEEE, Incline Village, 1–10.
[34] Jiguo Song, John Wittrock, and Gabriel Parmer. 2013. Predictable, efficient system-level fault tolerance in Cˆ 3. In *2013 IEEE 34th Real-Time Systems Symposium*. IEEE, Vancouver, 21–32.
[35] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1626–1629.
[36] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, Snowbird, 147–156.
[37] Elin Vinka. 2018. How many Zookeepers in a cluster? https://www.cloudkarafka.com/blog/2018-07-04-cloudkarafka-how-many-zookeepers-in-a-cluster.html. Accessed: 2020-10-01.
[38] Chao Wang, Christopher Gill, and Chenyang Lu. 2019. Frame: Fault tolerant and real-time messaging for edge computing. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Dallas, 976–985.
[39] Sisu Xi, Meng Xu, Chenyang Lu, Linh T. X. Phan, Christopher Gill, Oleg Sokolsky, and Insup Lee. 2014. Real-time multi-core virtual machine scheduling in Xen. In *2014 International Conference on Embedded Software (EMSOFT)*. ACM, New Delhi, 1–1.
[40] Meng Xu, Linh Thi, Xuan Phan, Hyon-Young Choi, and Insup Lee. 2017. vCAT: Dynamic cache management using CAT virtualization. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, Pittsburgh, 211–222.

Received April 2021; revised June 2021; accepted July 2021