ImmortalChopper: Real-Time and Resilient Distributed Transactions in the Edge-Cloud

Juncheng Fang, Farzad Habibi, Binbin Gu, Faisal Nawab
University of California, Irvine
{*junchf1, habibif, binbing, nawabf*}*@uci.edu*

***Abstract*—Emerging applications in the areas of real-time Internet of Things (IoT) and edge technologies require fast processing and response times. This motivates the utilization of edge nodes for storing and processing data close to the user. In settings with a vast number of edge nodes, the state of the data is distributed across a large number of edge nodes. This makes it expensive to perform distributed transactions as these transactions would span edge nodes that are connected via less reliable and relatively slow network infrastructure. It is prohibitive to use existing protocols like 2PC that require many rounds of communication across participants.**

In this paper, we propose *ImmortalChopper*, a distributed transaction processing protocol designed for the edge-cloud environment. The goal of *ImmortalChopper* is to provide *One-Node Response (1n-Response)*, a guarantee of transaction commitment by contacting only one node without waiting for coordination with the other nodes. To achieve this, we build on and extend the literature of transaction chopping and lazy replication. Transaction chopping breaks transactions into smaller *hops*. If the first hop commits, the rest of the transaction is guaranteed to commit, accomplishing the goal of 1n-Response. Each hop is replicated to tolerate temporary node failure, and we apply lazy replication on the first hop to maintain 1n-Response. However, combining transaction chopping and lazy replication without special care can lead to transactions operating on a stale state and potentially violating serializability. We present a new transaction chopping theory called *ChopperGraph* that integrates the notion of lazy replication and speculative execution. It ensures 1n-Response while preserving serializability. We evaluate *ImmortalChopper* on three applications and the result shows that it achieves 1n-Response in real-time and can quickly recover from node failure.

I. Introduction

Many emerging edge and Internet of Things (IoT) applications require fast response. This includes real-time IoT applications in smart cities and autonomous driving. To enable fast processing, it is necessary to allow users to store and process data on their own edge devices. This is already a common practice in many IoT and edge applications [1]–[4].

The ability of local processing also provides high availability for edge devices that are vulnerable to network outages. This can happen due to base station failures — a study [5] monitored around 800 base stations for 7 days and detected more than 3 thousand outage events. It is even more susceptible to edge nodes deployed in areas with poor signal coverage [6]. For example, an electric vehicle charging company requires real-time processing at all charging stations. However, some stations deployed in the wild may experience frequent network disconnections. If a cloud server is used, the driver cannot complete charging during a network failure and needs to wait at the station until recovery. Alternatively, deploying the application on the local edge nodes provides a consistent user experience for all stations and can tolerate network failures. We further illustrate this in Sec IV.

However, many edge applications require distributed transactions to coordinate between edge nodes. For example, the edge node in the charging station may need to contact other nodes to access user information. Ensuring the serializability of distributed transactions is also important for such applications — user information like balance should be consistent. Traditional distributed transaction protocols require multiple round-trips between the participants and cannot enjoy the benefits of local processing. Thus, they are not suitable in the edge-cloud environment.

To support distributed transactions and provide a user experience similar to local transactions, we propose our design goal called *One-Node Response (1n-Response)*. This is to ensure that the user will receive the transaction commitment guarantee by contacting a single node only—without that node communicating with other nodes first (The rest of the transaction and coordination with other nodes will be performed lazily in the background.) 1n-Response allows an edge node to respond to client requests in real-time for both local and distributed transactions, even when it is disconnected from other edge nodes.

There are two main problems to achieving the goal of 1n-Response: *(1) Distributed transaction coordination.* To ensure the atomic and serializable execution of distributed transactions, existing protocols like Two-Phase Commit (2PC) [7] require multiple rounds of synchronous coordination across participants for a transaction to commit. Traditional protocols cannot decide whether a distributed transaction can be committed without contacting all the participants. As a result, achieving 1n-Response is not possible with traditional protocols. *(2) Edge node failure.* Edge nodes are more susceptible to node failures and network disconnection compared to cloud nodes. Replication [8]–[10] is one of the most commonly used techniques for fault tolerance in distributed systems. However, replication violates the goal of 1n-Response since the node has to wait for the replication response before responding to the user.

In this paper, we propose *ImmortalChopper*, a distributed transaction protocol designed for the edge-cloud environments. It tackles both problems mentioned above to achieve 1n-Response. This is achieved by building on two main technologies: transaction chopping [11] and lazy replication [12]. Simply combining them will violate serializability, so we propose *ChopperGraph* to ensure the correctness of execution.

ImmortalChopper adopts transaction chopping for efficient distributed transaction processing in the edge-cloud. In the protocol, transactions are chopped into smaller pieces called hops. Each hop accesses objects in one edge node and is processed as a local transaction in that node. Each transaction is executed by invoking the hops sequentially, and the execution is serializable even without synchronous coordination across hops. Moreover, a transaction is guaranteed to be committed after the first hop of the transaction is executed successfully (i.e., contacting only the first edge node), which meets our goal of 1n-Response. To support the above features, the transactional workloads need to satisfy special theoretical properties (more details in section II).

We also utilize replication for fault tolerance. To maintain 1n-Response, we apply lazy replication on the first hop of each transaction. Lazy replication means that the node will send the replication message but will not wait for the response. As a result, the commitment guarantee can be sent to the user right after the first hop is processed in the edge node, and before communicating with the replication nodes. However, the replicated state in the backup node can be older than the state in the edge node when lazy replication is applied. So when an edge node fails, the replicated state is stale — therefore, reading from the replication nodes may violate serializability. The impact of stale data can go beyond that failed edge node since distributed transactions from other nodes may access the stale data in the replication node. This causes a cascading impact of stale data. The main challenge faced in this paper is that transaction chopping is not compatible with lazy replication. This is because the transaction chopping protocol must access the most recent state, while lazy replication may lead to reading a stale state.

We propose *ChopperGraph*, a *new transaction chopping theory that integrates the notion of lazy replication of the first hop.* The main innovation of ChopperGraph is that it models (1) the data dependencies between hops and (2) the type of hops. The data dependencies allow us to track the hops that have read stale state and mark them as "Non-Safe". Then, based on the dependencies, the hop types identify which hops can be executed speculatively and are safe to be performed, and which hops should be delayed (note that only hops other than the first hop can be delayed to ensure 1n-Response from first hops.) We integrate the ChopperGraph model and build the *ImmortalChopper*'s transaction processing protocol to ensure that we guarantee 1n-Response and serializability with both distributed transactions and lazy replication.

In summary, we make the following contributions:

- To the best of our knowledge, *ImmortalChopper* is the first distributed transaction protocol for edge-cloud applications that can achieve 1n-Response while supporting fault-tolerance and serializability. It enables distributed transactions to respond as fast as local transactions, even if the edge node is disconnected from the network.
- We proposed ChopperGraph, an extension of transaction chopping theory that identifies hops that can be executed speculatively and that need to be delayed in a failed node. It ensures 1n-Response when lazy replication is applied.
- We implemented a prototype of *ImmortalChopper* and conducted a comprehensive evaluation that compares with Transaction Chains, Two-Phase Locking, and Optimistic Concurrency Control baselines. The results show that 1n-Response has a significant impact on improving performance in terms of both latency and throughput. We also show that *ImmortalChopper* minimizes the impact of node failures and can recover efficiently.

II. Background

Transaction Chains [13] is a protocol designed to provide serializability with low latency, inspired by the idea of transaction chopping [11], [14]. It achieves low distributed transaction coordination costs and fast response when the workloads in the system satisfy specific requirements that we discuss later. In Transaction Chains, data tables are divided into shards by the row key, and each participant stores one shard. Then, each transaction is split into a sequence of hops T=$[h_1, h_2 ... h_n]$ where each hop is a local transaction that accesses data in one shard. The sequence of hops is called a *chain*. Transaction Chains provide two properties: (1) Executing a distributed transaction in a piece-wise (hop-by-hop) manner while ensuring serializability. Each hop only holds the locks of data items during the local (hop) transaction and early releases the locks before moving to the next hop. (2) Guaranteeing commitment of the whole transaction after the first hop of the transaction is processed. There are two stages of commitment: (1) commitment guarantee, which is a promise to the client that the transaction will be eventually committed, and (2) commitment completion, which means that the transaction is committed in all involved shards.

Example. Consider a transaction $T = R(X)R(Y)W(Y)W(Z)$ which accesses three different objects X, Y, and Z. It can be chopped into a chain with three hops $h1 = R(X)$, $h2 = R(Y)W(Y)$ and $h3 = W(Z)$. Each hop can be executed as a local transaction at the corresponding shard. To process the chain, the system will first send h1 to the shard that stores object X. The lock of object X is only held during the local transaction of hop h1. After h1 is processed, the commitment guarantee can be sent to the client. h2 and h3 are then executed one by one in the corresponding shards. The locks for h2 and h3 are only held during the processing of h2 and h3, respectively. After the last hop, h3, is processed, the commitment completion will be sent to the client.

Hop-by-hop execution cannot guarantee serializability with normal protocols and workloads. To ensure that chains are serializable as transactions, two types of properties need to be met: properties of the transaction processing protocol and properties of the transactions themselves.

- **Protocol properties:**
  - *Per-hop isolation*. Each hop is serializable with respect to other hops in all chains. This is achieved efficiently by executing a hop as a local transaction.
  - *Inner ordering*. Hop h_{i+1} never executes before hop h_i.
  - *All-or-nothing atomicity*. If the first hop of a chain commits, then the other hops should eventually commit. If the first hop aborts, then the whole chain should abort. Thus, the first hop determines the outcome of the chain.
- **Transaction properties:**
  - *SC-graph seriablizable*. The SC-graph of all involved chains must have no SC-cycles.

SC-Graph. In an SC-graph (see Figure 1), vertices are the transaction hops, and edges are the relations between hops. The vertices in the same chain are connected with (sibling) S-edges, while those in different chains that access the same data item with one of them being a write operation are connected with (conflict) C-edges. It has been proven that an SC-graph without SC-cycle (a cycle that

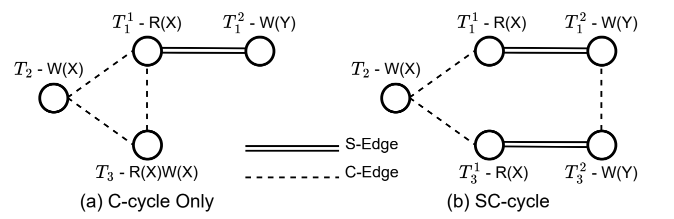


Fig. 1: SC-Graph analysis example. Graph (a) has a cycle with C-edge only, which is serializable. Graph (b) contains an SC-cycle that is not serializable.

includes both S-edges and C-edges) ensures serializability even with the aforementioned piece-wise execution [11], [14].

In Figure 1, there are three transactions T1, T2 and T3. Assume that T1 executes two operations, $R(X)W(Y)$ on two objects X and Y, which are located in different shards. We can chop T1 into T_1^1 for $R(X)$ and T_1^2 for $W(Y)$, where the two hops are connected with an S-edge. T2 executes $W(X)$, which has a read-write conflict with T_1^1, so a C-edge is added. There are two different scenarios for T3. (a) T3 is $R(X)W(X)$. Since it only operates on one object X, there is no need to chop T3. T3 has a conflict with T_1^1 and T2 which access object X. As a result, there is a C-cycle between T_1^1, T2 and T3. The chain is serializable because there is no SC-cycle. This means that any piece-wise execution of these transactions cannot lead to a serializability violation. (b) T3 is $R(X)W(Y)$. Now T3 will be chopped into T_3^1 and T_3^2 like T1. There is no C-edge between T_1^1 and T_3^1 since they are both read operations. However, a new C-edge for T_1^2 and T_3^2 is added which results in a cycle containing both C-edges and S-edges. So in this case, the chain is not serializable with piece-wise execution.

There are three limitations for Transaction Chains: (1) User-defined abort in the first hop only. (2) The set of chains should be predefined in order to perform the static SC-graph analysis. (3) The chains should be SC-Cycle free. *ImmortalChopper* inherits these limitations and we will discuss them more in Section III-H.

III. System Design

ImmortalChopper is a distributed transaction processing protocol designed for real-time and resilient distributed transactions in the edge-cloud environment. It achieves 1n-Response with the ChopperGraph model that combines transaction chopping and lazy replication. In this section, we first describe the system model and provide an overview of the system, then discuss the detailed protocols and algorithms.

A. System Model

System Components. *ImmortalChopper* consists of the following components (Figure 2a):

- *(1) Service manager node:* There is a single service manager node to coordinate server nodes. It is responsible for initiating server nodes when the system starts (Sec III-E) and updating the transactions chopping graph (Sec III-G). It also serves as a DNS server to host the addresses of server nodes and handle node relocation. Note that it does not participate in the transaction execution, so such a centralized node will not be a bottleneck of the system.
- *(2) Client and Edge nodes:* Each client hosts their edge node to store and process data that is most relevant to the client. Edge nodes are responsible for processing client requests. Data is partitioned across edge nodes. Each data item is stored on a single edge node (shared-nothing partitioning). An edge node typically stores data about clients who are close to it. The edge node processes client transactions that can be either local or they may be distributed and involve communicating with other edge nodes. *ImmortalChopper* edge nodes can be geo-distributed. Therefore, they may incur long wide-area latency when they communicate.
- *(3) Backup nodes:* The backup nodes are deployed in a highly available environment, typically on the cloud or a Paxos-replicated [10], [15] cluster. Each backup node is responsible for replicating the state of one of the edge servers. When the corresponding edge server fails, the backup node is utilized to process requests on the failed edge node's behalf until it recovers. Backup nodes are typically remote. Communication from a client or an edge server to the backup node would incur wide-area latency.

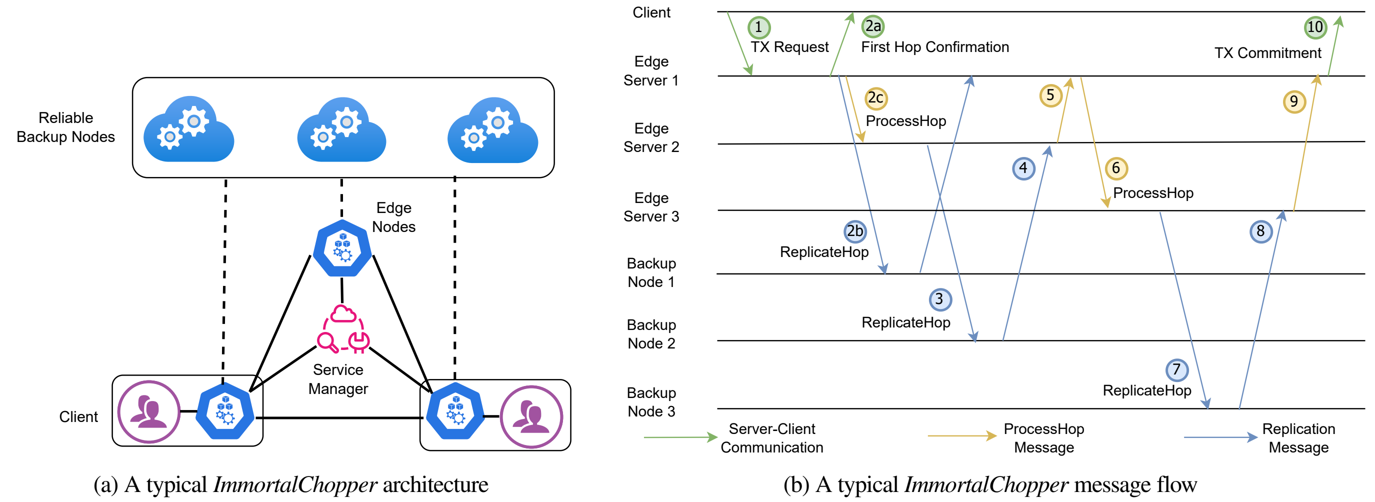
Data Model. Each transaction is chopped into smaller pieces called hops. The sequence of the hops is called a chain of the transaction. Each hop is executed as a local transaction that accesses data in one edge node. The piece-wise execution of hops in *ImmortalChopper* is only allowed for transactions that satisfy the acyclicity of the ChopperGraph (Section III-D). Therefore, it is required that the set of chains is predefined.

Failure Model. Edge nodes are more susceptible to temporary failure and network disconnection. For node failure, we assume that each node has a durable log to record all committed hops. Nodes can use RAID (Redundant Array of Independent Disks) to replicate logs locally, so the committed data will never be lost, and the node can recover eventually with the help of the logs. *ImmortalChopper* allows hops corresponding to the failed node to be executed speculatively and quickly recover to the consistent state after the node recovery. For network failure, it means that an edge node is disconnected from other edge nodes but it can still get the request from the clients. When an edge node is disconnected from the network, *ImmortalChopper* can still provide 1n-Response guarantee so that users don't need to wait for the network recovery.

B. Overview

ImmortalChopper is designed to achieve One-Node Response (1n-Response) on the edge. It provides a guarantee of transaction commitment by contacting only one node without waiting for coordination with the other nodes. There are two main challenges to achieve 1n-Response: (1) Traditional distributed protocols [16]–[18] have to communicate with all involved participants before responding to the client (2) Edge nodes are more vulnerable to failures compared to cloud nodes. Transactions that are already committed may be lost when a node fails. Replication is commonly used to ensure the durability of data and availability of the service, but it introduces extra latency and also violates 1n-Response.

We build on two pieces of literature to solve the above problems. Firstly, Transaction Chopping protocols divide distributed transactions into smaller pieces called hops. If the set of transactions



(a) A typical *ImmortalChopper* architecture (b) A typical *ImmortalChopper* message flow

Fig. 2: (a) shows an example architecture with three edge areas. There are three backup nodes deployed in the cloud. (b) shows an example of the message flow with a three-hop transaction running on three edge servers.

satisfies the acyclic SC-graph test [11], transactions can be executed piece-wise without distributed coordination. It also guarantees the successful and durable commitment of a transaction when the first hop is executed successfully, which only involves a single node. Secondly, we apply lazy replication on the first hops so that the response can be sent to the client without waiting for the replication acknowledgment. Note that other hops are still replicated synchronously to minimize the overhead of failure recovery. While this design choice slows down the full transaction completion time in the normal case, the 1n-Response guarantee ensures that users can still receive a real-time response.

However, transactions might see stale data when lazy replication is applied on transaction chopping (see details in Section III-C). Prior Transaction Chopping protocols assume that they access fresh data in all hops. Therefore, accessing stale data would violate their correctness. To overcome this challenge, we introduce *ChopperGraph* in section III-D, which extends transaction chopping theory to incorporate lazy replication and ensures 1n-Response.

Normal-Case Processing Overview. Figure 2b shows an example of *ImmortalChopper* on three edge servers. The scenario shows the execution of a transaction containing three hops, where the first hop data is stored in the closest edge server to the client. The client first sends a transaction request to the closest node, edge server 1, which serves as the coordinator of this transaction (step 1 in the figure). The coordinator drives the commitment of the transaction by performing the processing of the first hop and then sequentially coordinating with the other edge servers for the processing of the other hops. After the first hop is executed successfully, the coordinator dispatches three events concurrently: (2a) It sends the commitment guarantee message to the client, identifying that the request has been processed and that the transaction is guaranteed to be committed (1n-Response). (2b) It sends a ReplicateHop message to the corresponding backup node 1 to replicate the first hop lazily. (2c) It sends the ProcessHop message to edge server 2, which stores the second hop data of the transaction. The coordinator then waits for the response of 2c before it proceeds to the next step.

Edge server 2 processes the second hop after receiving the ProcessHop message. It then sends the ReplicateHop message to the corresponding backup node 2 and waits for the response (steps 3 and 4). Note that since this is not the first hop of the transaction, the replication is synchronous. We need to wait for the replication to be performed because the initial response is already sent to the client, and there is no need to bypass the synchronous replication of the later hops. Edge server 2 replies to edge server 1 after the replication is done. When the coordinator receives the response from edge server 2 (step 5), it sends the ProcessHop message to edge server 3 and waits for the response. The following steps 6–9 are the same as steps 2c–5 except for the corresponding nodes. Finally, the coordinator finds that all hops are processed. It sends the commitment completion message to the client (step 10).

As shown in the example, *ImmortalChopper* provides 1n-Response by sending the commitment guarantee to the client once the first hop is processed successfully, without waiting for it to be replicated or the later hops to be processed. In the rest of this section, we show the details of *ImmortalChopper* protocol and how it maintains serializability despite failures.

C. Challenges of Edge Failures

As discussed previously, applying lazy replication on the first hop can lead to some transactions accessing stale state. We provide the details of the challenges ensuing from node failures when combining transaction chopping and lazy replication in this subsection. This is the motivation of our ChopperGraph and *ImmortalChopper* proposals.

To describe the challenges of edge failures, consider the example in Figure 3. The figure shows a timeline of how logs in each node are stored and replicated. At step 1, edge node 1 receives a ProcessHop request for transaction 1 hop 2 (denoted as T1H2) that writes 1 to x. This request is kept in the local log and is also replicated to backup node 1. At step 2, node 1 receives a request for T2H1 that writes 2 to x. Since T2H1 is the first hop of a transaction, the commitment guarantee is sent to the client concurrently with the replication message to the backup node. During this step, Node 1 fails, and the replication message is dropped. However, the user may have received the guarantee (it is important because this transaction

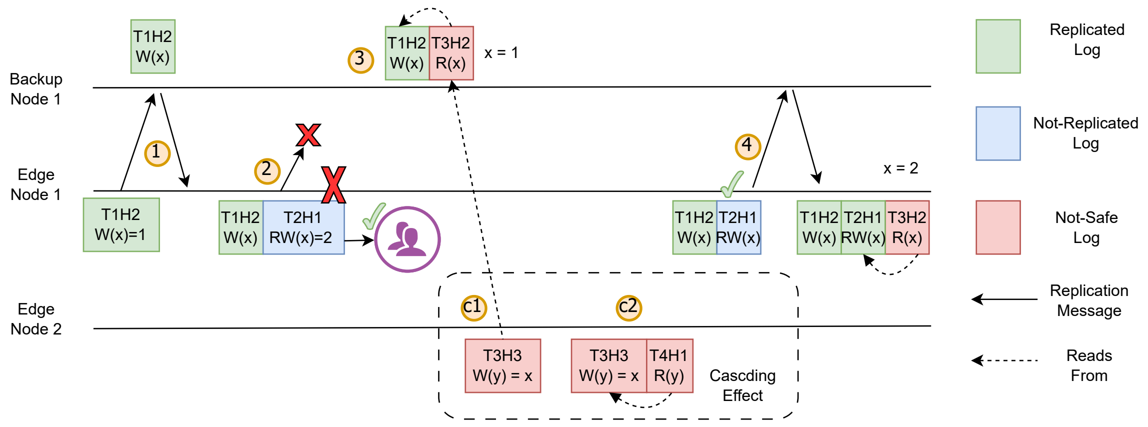


Fig. 3: An example of a timeline of logs during an edge node failure

cannot be rolled back if the client received the guarantee). Then at step 3, a request for T3H2 (T3H1 is already processed in another node) cannot reach node 1 so it is redirected to backup node 1. T3H2 reads the value of x as 1 because backup node 1 did not receive the log of T2H1 that modifies x to 2. Finally (c1 and c2 will be discussed later), at step 4, node 1 recovers and retrieves the new logs from the backup node. Node 1 appends the new logs to the end of the local logs and reprocesses the new entries. It finds that the result of T3H2 is changed from 1 to 2, meaning T3H2 was processed incorrectly on stale state. We cannot overwrite the local log with the replicated log because the commitment guarantee of T2 might have been sent back to the client — i.e., discarding T2H1 violates 1n-Response. As a result, all hops performed on the backup node are considered Non-Safe because they might see a stale state. They cannot be committed until the edge node recovers.

Moreover, there can be a cascading effect on the hops that are processed on the other edge nodes. Suppose that T3H3 is a hop that depends on the output of T3H2, which is Non-Safe. It is processed and logged in node 2 at step c1 (before the edge node 1 recovers at step 4). Consequently, T3H3 is also Non-Safe because it writes y with stale information about x from T3H2. Later at step c2, a request for T4H1 comes and reads the value of the stale y. T4H1 cannot be safely committed before the value of y is confirmed — this violates the 1n-Response. We need to ensure that first hops always operate on the most recent state. So, Non-Safe hops that can affect first hops (like T3H3) need to be delayed until they can be safely committed. To track all Non-Safe hops and delay hops that can affect first hops, we introduce a new transaction chopping theory called ChopperGraph in the next section.

Allowing speculative hop execution in the backup node makes the system design more complicated. We make this decision instead of pausing a hop when the corresponding node fails based on two reasons: (1) There can be a large number of paused requests waiting for the failed node — they will flood into the node once it recovers, which can lead to overloading the node. It causes further performance-related failures, such as metastable failures [19], [20]. (2) The number of logs that are processed in the failed node but not replicated to the backup node (like T2H1) is typically low. Most of the Non-Safe hops are processed using the correct state. This enables us to incorporate their changes without the need for reprocessing them. Therefore, we allow useful work about the failed node to be performed and then verify their safety after the recovery. This can significantly reduce the time for system recovery.

D. *ChopperGraph*

In this section, we describe the proposed transaction chopping theory called ChopperGraph. It identifies hops that read stale states and determines if those hops need to be delayed.

Definition 3.1: We define the ChopperGraph (ChG=(V,E)) as the following: Each vertex $v \in V$ represents a hop of a transaction. Vertex v_i^j corresponds to the j^{th} hop in transaction t_i, denoted h_i^j. There are the following types of edges:

- Sibling edge (s-edge): Between any two vertices v_i^j and $v_i^{j'}$ in the same transaction t_i, there exists an s-edge.
- Conflict edge (c-edge): Between any two vertices v_i^j and v_k^l, there exists a c-edge if (1) They are from difference transactions (i.e., $i \neq k$), and (2) They access the same data item, x, and at least one of them performs a write on x.
- Directed Dependency edge (d-edge): Consider a transaction t_i. There exists a directed d-edge from v_i^j to v_i^k if (1) $j < k$, (2) h_i^j has a read operation, and (3) h_i^k has a write operation. This represents that hop h_i^k performs a write that depends on the output of v_i^j.

Vertices in the ChopperGraph can be classified as one of the following types (see Figure 4 for examples):

- First-hop (1^{st}-*hop*): A vertex that represents the first hop of a transaction. 1^{st}-*hop* always processes the most recent data on the edge node. Once a 1^{st}-*hop* is invoked, it cannot be rolled back or reordered due to the 1n-Response property.
- Unorderable-hop (*u-hop*): A *u-hop* vertex is a hop h_i^j that (1) has a path of c-edges to a 1^{st}-*hop* vertex, and (2) is a destination of a d-edge—meaning there is a d-edge from h_i^k to h_i^j, where h_i^k is not a 1^{st}-*hop* vertex. A *u-hop* cannot be reordered (i.e., reprocessed) as their processing may potentially impact a 1^{st}-*hop* that cannot process on stale state.
- Orderable-hop (*o-hop*): A hop that is not a 1^{st}-*hop* nor a *u-hop*. An o-hop can be processed speculatively as they do not influence a 1^{st}-*hop*.

ChopperGraph extends the original SC-Graph by adding a new type of edge called the directed dependency edge (d-edge) and by classifying hops into three categories. D-edge helps analyze data

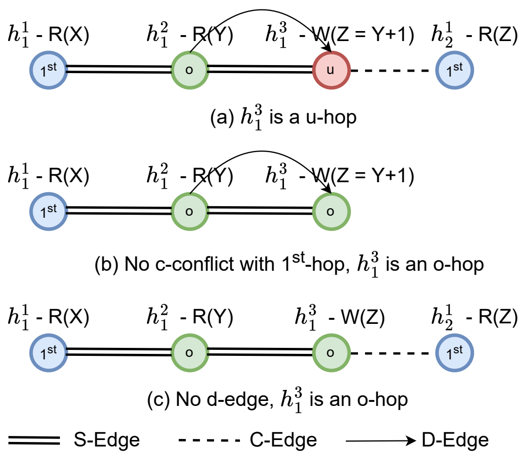


Fig. 4: ChopperGraph examples. Graph (a) shows h_1^3 is a u-hop since it has a c-conflict with a 1^{st}-hop and is the destination of a d-edge from h_1^2. Graphs (b) and (c) show that h_1^3 is an o-hop when any of the two conditions is not satisfied.

dependencies within a transaction. This type of edge is added when there is a latter hop (dest) that relies on the reads of the earlier hop (source). When the corresponding edge server of a source hop fails, the source hop is processed by the backup node, so it is marked as Non-Safe. The dest hop of the d-edge will also need to be marked as Non-Safe since it reads the stale data from the source hop.

ChopperGraph's three types of hops have different behaviors when there is a failure. (1) First-hop: 1^{st}-hop always processes on the most up-to-date state to ensure the correctness of the fast commitment guarantee. Therefore, it cannot be processed by the backup node since the most recent state might be in the edge node. (2) Unorderable-hop: when the source hop of the u-hop is marked as Non-Safe, it cannot be processed until all its source hops are committed. This is because the u-hop can affect the first-hop of other transactions, which breaks the guarantee of 1n-Response when the u-hop is Non-Safe. (3) Orderable-hop: o-hops are the most common hops that can be processed speculatively. It helps reduce the recovery time since most hops don't need to be reprocessed.

Wait dependencies during failure. As we discussed in Section III-C, hops will be marked as Non-Safe if they are processed by the backup node when the corresponding edge server fails. Other hops that depend on those hops will also be affected. Thus, they are also considered Non-Safe and can be committed after all the dependent hops are committed. We call this type of dependency during a failure as a *wait dependency* as follows:

Definition 3.2: In an instance of ChG, there can be two types of wait dependencies:

1. Intra-shard dependency: this is a dependency from one hop h_i^k to h_j^l, where h_j^l waits for h_i^k to be fully committed. The dependency exists if h_i^k writes an object x that h_j^l reads. (Since there is a c-edge between the two hops, they are both on the same shard).
2. Inter-shard dependency: this is a dependency from hop h_i^k to h_i^l of the same transaction, and there exists a d-edge from v_i^k to v_i^l. Hop h_i^l waits for h_i^k to be fully committed.

Wait dependencies are used to track the cascading Non-Safe hops during failure. *ImmortalChopper* can follow the dependencies to resolve those hops after the failed node recovers.

Correctness Proof Sketch. We show that if there is no SC-cycle in ChopperGraph, transactions can be executed piece-wise while ensuring serializability even with edge node failures. We prove this by contradiction. Assume that there is a serializability conflict in *ImmortalChopper* for transactions that form a ChopperGraph with no SC-Cycles. This means that there is a sequence of transactions that would lead to a cycle in the serializability graph [7], *i.e.*, $t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow t_1$. We expand this cycle into the hops that form each transaction to be: $h_1^a \xrightarrow{c} h_2^b \xrightarrow{c} \ldots \xrightarrow{c} h_1^b \overset{s}{\sim} h_1^a$, which indicates the existence of an SC-Cycle in ChopperGraph. The symbols $\overset{s}{\sim}$ and $\xrightarrow{c}$ indicate sibling and conflict edges, respectively. Given that ChopperGraph contains no SC-Cycles, it is impossible that the cycle in the serializability graph can form [11].

The above is guaranteed when execution is carried out by edge nodes without failures. In the case when a failure of some edge node exists, there is the additional complexity of hops that are temporarily impacted by stale states. We need to prove that if a transaction reaches commit completion, then all its stale reads are corrected. Assume to the contrary that a hop h_i^j was initially performed by the backup node and thus has read stale state. The stale state might impact the hop itself as well as other hops that depend on it. However, by our tracking of dependencies, we can guarantee that such stale state is not impacting transactions that reached the commitment completion state. A stale state of hop h_i^j may impact: (1) the hop itself: this is rectified by the edge node during recovery (Section III-F), (2) a hop, $\mathcal{H}$, in the same transaction that reads from h_i^j: this is rectified by enforcing restarting $\mathcal{H}$ after hop h_i^j's recovery with intra-shard dependencies. (3) a hop, $\mathcal{H}$, in another transaction that depends on h_i^j or its dependents: this is rectified by enforcing restarting $\mathcal{H}$ due to inter-shard dependencies. Finally, it is always possible to restart any such hop. If there exists a dependent hop $\mathcal{H}$ that cannot be restarted (i.e., a first hop), then h_i^j would be marked as an Unorderable-hop and would not be executed speculatively on the backup node. With the conditions above, it is a contradiction that a serializability conflict may occur or that a transaction that has reached commitment completion has read from a stale state.

E. System Initiliazation

Chains need to be set up before the system starts operating requests. Each hop is implemented as a stored procedure. The service manager node builds the ChopperGraph and performs a static analysis on the graph. To construct S-edges, we connect the hops in the same chain. When constructing C-edge, the system cannot know which data item (row) will be accessed because each hop only defines the table and column to be operated on. As a result, we add C-edges between hops from different chains that access the same column, and at least one of them contains a write operation. There might be some C-edges that are not necessary, so we allow the programmers to mark edges as commutative. The system will ignore commutative edges when checking for SC-cycles. For D-edges, we check the output of each hop and connect the later hops that use the output as their input parameters. After the edges are constructed, we iterate through the vertices to mark their vertex types. A real-world application example of how to define chains and build ChopperGraph from a set of transactions is shown in Section IV.

Algorithm 1 Coordinator: BeginTransaction event

```
1: $i$ := new txnId
2: $t_i$ := the received transaction
3: Process and log $h_i^1$
4: if $h_i^1$ aborts then
5:     Reply the Abort message to the client
6: else
7:     [Async] Send Commitment Guarantee message to the client
8:     [Async] Send ReplicateHop message to the backup node
9:     waitlist := empty list
10:    for $j \leftarrow 2$ to $N$ do                    ▷ The chain has N hops
11:        isSafe := DispatchHop($h_i^j$)
12:        if !isSafe then
13:            for $h_i^k$ in $h_i^j$.dependents do
14:                Add $h_i^j$ to the inter-shard dependency of $h_i^k$
15:            append $h_i^j$ to waitlist
16:    while waitlist is not empty do
17:        $h_i^j$ := received HopCommitted message
18:        remove $h_i^j$ from waitlist
19:        for $h_i^k$ in $h_i^j$.dependents do
20:            remove $h_i^j$ from the inter-shard dependency of $h_i^k$
21:            update the input of $h_i^k$
22:            if Inter-shard dependency of $h_i^k$ is empty then
23:                [Async] Send RemoveDP to the edge node of $h_i^k$
24:    Reply Commitment Completion message to the client
```

After the ChopperGraph of the application is built, the service manager checks whether the graph contains SC-cycles. If there are no SC-cycles, the chains can be executed piece-wise with *ImmortalChopper* algorithms (Section III-F) while ensuring the serializability of the transactions. Otherwise, they will be executed as normal distributed transactions that cannot enjoy the benefits of *ImmortalChopper*. The service manager then invokes all server nodes and propagates the ChopperGraph to them.

F. Design Details

In this subsection, we describe *ImmortalChopper*'s detailed algorithms for each system component.

Coordinator. To initiate a transaction, the *client* sends a BeginTransaction request with the *chainId* and the corresponding parameters of the chain to the edge node that stores the first hop data. The coordinator assigns a *txnId* to the request. It starts processing the first hop of the request by invoking the stored procedure as shown in Algorithm 1 (lines 1–3). If the first hop aborts due to a user-defined check, the transaction is aborted. The Abort message is then sent back to the client (lines 4–5).

If the first hop is processed successfully, two messages will be sent asynchronously (lines 7–8) and the coordinator will not wait for their responses: (1) The commitment guarantee message is sent to the client, indicating the request has been processed and that the other hops are guaranteed to be committed (thus achieving 1n-Response). (2) The ReplicateHop message is sent to the backup node of the coordinator to replicate the state (Note that in all algorithms, we will not wait for the response of a message if we put [Async] at the beginning). After sending the messages, the coordinator starts to sequentially dispatch the other hops of the transaction to their corresponding edge nodes (lines 9–15). In the DispatchHop function, messages will be redirected to the backup node if the edge node fails to respond within a timeout. If the response indicates that hop h_i^j is Non-Safe, the coordinator adds h_i^j to the inter-shard wait dependency of its dependent hops (i.e., hops that depend on the output of h_i^j) according to the d-edge. A hop is Non-Safe if it has any inter/intra-shard wait dependency or when it is processed by the backup node as discussed in Section III-D. Hops that are Non-Safe will also be appended to the waitlist.

Algorithm 2 Edge Node: ProcessHop event

```
1: $h_i^j$ := the hop to be processed
2: if $h_i^j$ is a u-hop and $h_i^j$.interDP is not empty then
3:     wait until $h_i^j$.interDP is empty
4: writeReadDpendencies := Process and log $h_i^j$
5: for $h_k^l$ in writeReadDpendencies do
6:     if $h_k^l$ is Non-Safe then
7:         append $h_k^l$ to the intra-shard wait dependency of $h_i^j$
8: response := send ReplicateHop message to the backup node.
9: isSafe := $h_i^j$.interDP is empty and $h_i^j$.intraDP is empty
10: Reply isSafe to the coordinator
```

Algorithm 3 Edge Node: Recover event

```
1: Pull the logs from the backup
2: for each new log l do
3:     Apply log l
4:     if l.hop has no wait dependency then
5:         [Async] Send HopCommitted to the coordinator of l
6: Replicate all logs to backup
```

After all hops are processed, the coordinator waits for hops that are Non-Safe to be resolved, as shown in Algorithm 1 (lines 16–23). When a Non-Safe hop is safely committed in the edge node, a HopCommitted message will be sent to the coordinator. The coordinator removes the committed hop h_i^j from the waitlist. It also removes h_i^j from the inter-shard wait dependency of the dependent hops of h_i^j and updates the input of those hops. If a dependent hop h_i^k has no other inter-shard wait dependency after the removal, the coordinator sends a RemoveDP message to the corresponding edge node to check if the hop can be safely committed. The edge node will reprocess hop h_i^k if the input has changed and check if the hop can be safely committed.

Finally, all hops are safely committed when there is no Non-Safe hop inside the waitlist. The coordinator replies with a transaction commitment completion message to the client.

Edge Node. There are two types of events for an edge node: (1) processing a hop and (2) handling recovery.

Algorithm 2 describes the processing of a ProcessHop event. The Processhop message from the coordinator includes the information of the hop h_i^j to be processed. The edge node first checks if h_i^j is a u-hop and if it is waiting for any other hops to be committed (lines 2–3). A u-hop cannot be executed until its source hops are all safely committed, as discussed in Section III-D. This is to ensure that first hops always process on the most up-to-date state to achieve 1n-Response. Then, the edge node executes the stored procedure. For each item h_i^j reads, we record the hops that perform the last write on the items and put them in the list writeReadDepencies. If any hop in the list is Non-Safe, we add the hop to the intra-shard wait dependency of h_i^j (lines 5–7). After that, the edge node sends the ReplicateHop message to the backup node and waits for h_i^j to be replicated. Finally, it checks if h_i^j has any wait dependency and replies to the coordinator accordingly.

When an edge node recovers from a failure, it pulls all logs from the backup node, as shown in Algorithm 3. It then processes the

Algorithm 4 Edge Node: RemoveDP event

```
1: if Invoked by coordinator then
2:     set $h_i^j$.interDP to empty
3: if $h_i^j$ has no wait-dependency then                ▷ $h_i^j$ is safe
4:     Reprocess hop $h_i^j$ if input is changed
5:     [Async] Send HopCommitted to the coordinator of $h_i^j$
6:     for all $h_k^l$ that has intra-shard dependency with $h_i^j$ do
7:         remove $h_i^j$ from the intra-shard dependency of $h_k^l$
8:         if $h_k^l$ has no intra-shard dependency then
9:             invoke RemoveDP($h_k^l$)
```

new logs that are only in the backup node. If the corresponding hop of a log has no wait dependencies, that hop is now safely committed. The edge node then sends a HopCommitted message to the coordinator of the transaction. Finally, the new state of the edge node is replicated to the backup node.

Note that the system is not fully recovered after the Recover event of the failed edge node is done. There are still hops in other edge nodes that are waiting for the recovery of the failed node (due to inter-shard wait dependency). The coordinator serves as the bridge to resolve the inter-shard dependency: when the coordinator receives a HopCommitted message, it removes the committed hop from the inter-shard dependency list of the dependent hops. It sends a RemoveDP message to the edge node if any of the dependents has an empty list (Algorithm 1 lines 16–23).

Algorithm 4 shows the process of removing the wait dependency of a hop. When the RemoveDP event is invoked by the remote coordinator, it means that the inter-shard dependency of h_i^j is now empty so the edge node should also empty the list (lines 1–2). If h_i^j has no wait-dependency, the edge node will reprocess the hop only if the input is changed. It then send the HopCommitted message to the coordinator (lines 4–5). For any h_k^l that waits for h_i^j, it removes h_i^j from the intra-shard dependency of h_k^l (lines 6–9). Furthermore, if h_k^l has no intra-shard dependency after the removal, the RemoveDP event on h_k^l will be invoked recursively to resolve the chain of dependencies (lines 8–9). Note that each hop will be invoked in RemoveDP at most twice, once from the coordinator to remove the inter-shard dependency and once from the edge node itself to indicate the intra-shard dependency is empty.

Backup Node. When the edge node is operational, backup nodes are only responsible for replicating the logs. However, if an edge node fails, the backup node will take the responsibility of handling the ProcessHop messages from the coordinator. The process is similar to Algorithm 2 in the edge node. The two main changes are: (1) No need to wait for u-hops since 1^{st}-hop can only be processed by the edge node. As a result, u-hops processed on the backup node will not affect 1^{st}-hops. (2) Always reply Non-Safe, since the backup node might not have the most up-to-date state.

G. Updating ChopperGraph

While the chains of transactions are defined at the start of the system, *ImmortalChopper* also allows updating the ChopperGraph on the fly. The updated transactions are sent to the service manager node, and it generates the new ChopperGraph. Most transactions are not affected by the update, which means their hop processing code is not changed (the transaction itself is not updated), and no new edges are connected to them (the updated transactions do not violate the serializability of the transaction). Those transactions can run on the edge nodes without being interrupted by the update. To ensure that the updated transactions are consistent across all edge nodes, the service manager runs a three-phase protocol: 1) Prepare phase: the service manager broadcasts a prepare message with the list of the updated transaction IDs. When the edge node receives the update, it stops accepting new transaction requests that are in the list of updated transactions. After all existing requests in the list of updated transactions are finished, the edge node replies with an ACK (Acknowledgement) to the service manager. This phase ensures that no request related to the updated transactions remains in the system. 2) Update Phase: After the service manager receives ACKs from all edge nodes, it broadcasts the updated ChopperGraph. Each edge node installs the updated ChopperGraph and replies with an ACK. This phase ensures that all edge nodes have the latest view of the ChopperGraph. 3)Complete Phase: After the service manager receives ACKs from all edge nodes, it broadcasts the completion message. Once the edge node receives the completion message, it can start accepting requests for the updated transactions.

H. Limitations

In this subsection, we discuss a few limitations of *ImmortalChopper* on the types of applications that can fully utilize the benefits and the availability of the service.

Limits on the applications. *ImmortalChopper* supports specific applications that have a set of predefined transactions. A recent survey [21] has shown that in real-world applications, many transactions are predefined where the read/write set can be inferred by analysing the code in advance or can be converted to such a format with minimal changes.

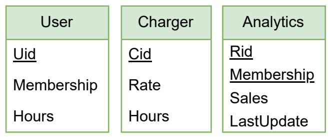
To execute the transactions piece-wise, the constructed ChopperGraph should not contain any cycles. If an application does not meet the requirement, *ImmortalChopper* will execute the transactions with traditional distributed transaction protocols like two-phase commit. Also, user-defined abort can only be defined in the first-hop. Otherwise, we cannot determine if the transaction can be committed by contacting only the first edge node.

We use TPC-C benchmark as an example: all transactions are pre-defined and have no user-defined abort. In terms of SC-Cycle, the new-order transaction consists of multiple updates to the stock level, which results in a potential conflict when requesting from a remote stock. The delivery and payment transaction are SC-cycle free.

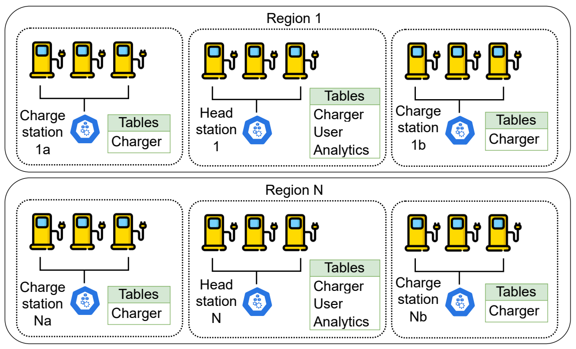
Failure of coordinator node. When there is an edge node failure, clients can not initiate transactions that start from the failed node. This is because the first hop can not be executed on the backup node, which doesn't have the up-to-date states. Typically, this is not a problem since the client is often co-located with the closest edge node (the client hosts the edge node itself), and the first hop is often executed in that node (due to data locality). So when the edge node fails, the client also fails and cannot send any requests. Also note that if the edge node is disconnected from the other edge nodes, instead of failing, all transactions can still be processed with the 1n-Response guarantee.

IV. Usecase Analysis

In this section, we will discuss how an application can use *ImmortalChopper* as the database engine using an example of an



(a) Schema



(b) Partition

Fig. 5: Schema and data partition of the charging application.

electric vehicle (EV) charging application.

A large chain EV charging company wants to keep track of the sales in different regions and from different membership levels of customers. There are a few charging stations and a head station in each region. The table schema is shown in Figure 5a, which contains three tables: User, Charger, and Analytics. The user table stores the user ID, membership level (user tier), and total charging hours. The charger table stores the charger ID, hourly rate, and total operating hours. Lastly, the analytics table stores the total sales and last update time of each membership level in each region ID.

To ensure the best experience for all users, the company requires real-time response in any region. Cloud databases are not feasible because regions far away from the cloud might suffer from high delays. Meanwhile, some charging stations will be deployed in areas with poor signal coverage, which means they might be disconnected. As a result, geo-distributed databases are also not a good fit because the chargers cannot connect to the database when the network fails. The customers will need to wait until the network recovers, which is a bad user experience. Finally, the edge-cloud model is adopted, where the edge nodes are collocated with the chargers. Even when the charging station is disconnected from the network, customers can still get real-time responses from the local edge node. Each charging station holds an edge node to store part of the data (see Figure 5b). It serves the requests from the chargers. The charger table is partitioned so that each edge node stores the information of chargers in the corresponding charging station. The head station stores the analytics table of that region, the partition key is *RegionId*. Customers will register their accounts at a head station. Their user records will be stored in the edge node of that head station.

The application supports two types of transactions, *Charge* and *ReadSales*. The charger will initiate the *Charge* request to the edge node in the station when a customer finishes charging at that charger. The input of a *Charge* transaction is the *ChargerId*, *Hour*, *UserId*, and *RegionId*. The transaction first reads the charger table to get the hourly rate and also increases the total operating hours of the charger. Then, it accesses the user table to increase the total hours of the user and read the membership level of the user. Finally, the analytics sales record of the specific *RegionId* and membership level is updated. The charging price is added to the sales, and the last update time will be recorded. This transaction can be chopped into three hops—h_c, h_u, and h_a where each hop accesses only one table as shown in Figure 6. The *ReadSales* transaction reads the region analytics table to get the sales record of all membership levels in a certain *RegionId*. It accesses multiple records under the same *RegionId*, so they are all stored in the same server. As a result, the *ReadSales* chain doesn't need to be chopped, and there is only one hop h_r.

The ChopperGraph (ChG) is constructed after defining the chains. As shown in Figure 6, the graph contains 3 chain instances—2 for the *Charge* chain and 1 for the read-only *ReadSales* chain. S-edges are added to connect the three hops h_c, h_u, and h_a of the *Charge* chain sequentially. Each hop of the *Charge* chain contains a write operation, so there are c-conflicts with the same hops of the other instance. The only hop h_r of *ReadSales* chain is a read-only hop that reads the Analytics table, so it has c-edges with h_a, h'_a.

The full ChG contains multiple SC-Cycles. However, two pairs of c-edges can be removed with careful analysis. For the c-edge between hop h_c and h'_t, the conflict is caused by the increment operation on the charger's total hours, which is a commutative operation with no conflicts. Similarly, the c-edge between hop h_u and h'_u can be removed. By deleting the two removable c-edges, the ChG has no SC-cycle.

The input of h_a relies on the output of h_c and h_u — the ticket price and user membership level, so the d-edges from h_c and h_u to h_a are added. Since h_a also has a c-edge with h_r which is the first hop of *ReadSales* chain, it is marked as an unorderable hop. As a result, h_a won't be executed until h_u has persisted on both the server and the backup service. It guarantees that h_r will never read data that can be rolled back.

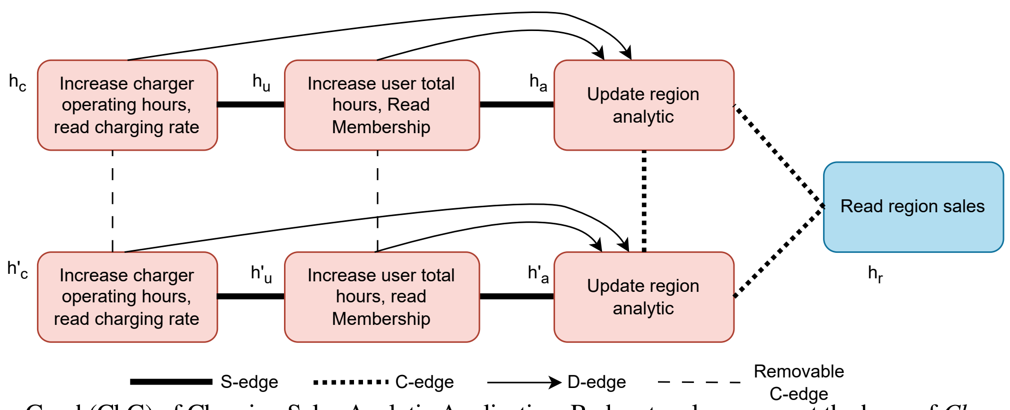
After removing unnecessary c-edges, the application has no SC-cycle so the transactions can be executed piece-wise with *ImmortalChopper*. It allows chargers to get the commitment guarantee of *Charge* in real-time. The unorderable hop is identified so the system can ensure that market analysts always get the correct sales statistics, even when there are node failures.

V. Evaluation

To validate the design of *ImmortalChopper*, our experiments focus on two key aspects: 1) In terms of performance, can *ImmortalChopper* achieve 1n-Response in real-time? How does *ImmortalChopper*'s performance compare with existing distributed transaction commit protocols under different environment settings? 2) In terms of fault tolerance, can *ImmortalChopper* minimize the impact of failures and recovers efficiently?

A. Implementation

Communication, Storage, and Transaction Implementation. The prototype of *ImmortalChopper* is implemented with the Go Language. The backup service is implemented as a variant of the server, and we assume it is highly available during the experiment. The definition of chains, including the number of hops and the function of each hop, is written as predefined transaction procedures and is loaded when the system starts.



h_c
Increase charger operating hours, read charging rate
h_u
Increase user total hours, Read Membership
h_a
Update region analytic
Read region sales
h_r
h'_c
Increase charger operating hours, read charging rate
h'_u
Increase user total hours, read Membership
h'_a
Update region analytic
S-edge
C-edge
D-edge
Removable C-edge

Fig. 6: The ChopperGraph(ChG) of Charging Sales Analytic Application. Red rectangles represent the hops of *Charge* chain while blue ones are for the *ReadSales* chain. There are multiple SC-Cycles initially, but they can be eliminated after removing unnecessary c-edges.

Comparison Baselines. We implement 2PL [22], OCC [23] and Transaction Chains [13] protocols to serve as comparison baselines with *ImmortalChopper*. For the 2PL implementation, we assume that all items to be locked are known before running the transaction. The client first runs an execution phase that acquires locks on all accessed items in the servers. After all the locks are acquired, the client executes the commit phase to apply the buffered write items and releases all locks. For OCC, we implement a distributed OCC protocol. The client first runs a read phase that puts shared locks on all read items. Then it executes the validation phase, which attempts to acquire exclusive locks on the write items and checks if all shared locks are still acquired by the transaction. The exclusive lock can invalidate the shared lock. The transaction will be rolled back if the validation phase fails. Otherwise, the write phase will be executed to release all locks and persist the writes in storage.

Note that for all three baselines, logs in edge servers are replicated to the backup service. For Transaction Chains, logs are replicated synchronously per hop (no lazy replication on the first hop). For 2PL and OCC, the replication is performed once per phase, similar to Paxos commit [24] and Spanner [17]. For example, the lock information is replicated after the lock phase, and the unlock (commit) is replicated after the commit phase for 2PL.

B. Experimental Setup

The experiments are conducted on skylake nodes of Chameleon Cloud [25]. Each machine has 2 Intel Xeon Gold 6126 CPUs at 2.6 GHz with 24 virtual threads. Each node is also equipped with 192GB RAM and 10 Gigabit Ethernet.

Unless otherwise stated, five Docker containers are running to mimic different geographic areas and services. Each container is assigned a limited number of virtual threads (i.e., 8 threads) to simulate edge nodes with limited computing power. The first four containers represent the application users in four edge areas, where each edge area has two *ImmortalChopper* servers and one client machine that issues transactions representing many clients. The fifth container is for the backup service, and it runs eight backup nodes, one for each edge server. The network round-trip time (RTT) is the following: 1ms for communication within the same edge area, 40ms for communication across edge areas (approximately half of the US West-to-East average RTT), and 60ms for edge-to-cloud communication between an edge node and a backup node.

We evaluate three different applications: one application is evaluated in depth to show the performance of *ImmortalChopper*, the other two are discussed briefly in section V-G to show the applicability. For the first application, we implement and extend the application described in Section IV. All three tables are evenly partitioned by rows across the eight servers. Each client service runs 20 concurrent threads for sending requests by default. The request parameters are distributed such that the ChargerId and RegionId are always partitioned on the two servers in the same edge area as the client, which means that the first hop and third hop are executed in the closest servers (This aims to mimic data locality). The UserIds are evenly distributed among all servers. The number of membership levels controls the level of contention in the *Charge* transaction (i.e., the source of conflicts in the third hop). By default, the number of membership levels is set to 50, so the data contention level is moderate. We also added local transactions that have no conflicts with the *Charge* chain, like modifying the user profile.

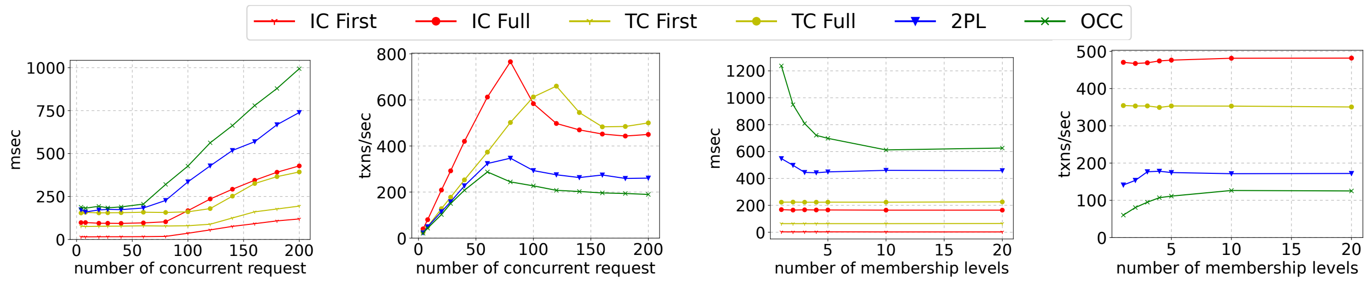
In the rest of this section, we present the evaluation results with different configurations. *ImmortalChopper* and Transaction Chains are abbreviated as IC and TC, respectively. The suffix *"First"* represents the commitment guarantee latency, and *"Full"* indicates the commitment completion latency.

C. Varying Concurrency

In this subsection, we evaluate the performance of *ImmortalChopper* under different workloads by varying the number of concurrent requests in the system. The distributed transactions ratio is set to 50%. Figure 7a shows the average latency, and Figure 7b shows the average transaction throughput. The x-axis represents the number of concurrent requests in the whole system, so 100 concurrent requests mean that each client node maintains 25 concurrent requests.

With a light workload (low number of concurrent requests), all protocols scale well because the computing and network resources are not congested. OCC gets its best performance at 60 concurrent requests with 287.39 txns/sec, while 2PL has a higher peak with 347.05 txns/sec. This means that OCC does not reduce contention compared to 2PL, but still suffers from the consequences of aborting due to failed validations.

IC scales up to 80 concurrent requests and reaches the highest throughput with 765.77 txns/sec. Meanwhile, TC reaches the peak of 659.91 txns/sec later at 120 concurrent requests. They



(a) Average Latency (b) Throughput (c) Average Latency (d) Throughput

Fig. 7: (a) and (b) show the performance when varying the number of concurrent requests. (c) and (d) vary the data contention levels. *First* represents commitment guarantee while *Full* represents commitment completion.

outperform 2PL and OCC because there are fewer messages and communications for the distributed transaction state management. The average first-hop latency of IC is only 10 msec because the first hop is processed by the closest servers within the same area. It shows that clients can get the commitment guarantee of the transaction immediately without waiting for the whole transaction to be committed (Fast 1n-Response). IC First is also faster than TC First since the guarantee can be sent before the replication is done. The difference is much more significant in real applications because the replication latency is typically much higher than 60 ms.

After the peak performance, TC has a higher throughput than IC. This is because the bottleneck of the system shifts from the network to the CPU when the number of concurrent requests grows. While asynchronous replication helps hide the network latency, it stresses the CPU more rapidly (hop processing is submitted at a higher rate) and may lead to lower performance under CPU-intensive workloads. Note that in terms of user experience, IC still outperforms TC since the commitment guarantee latency is the lowest.

D. *Varying Data Contention*

In this subsection, we run experiments to show that *ImmortalChopper* is stable under different levels of data contention. All transactions are distributed transactions. Figure 7c shows the average latency, and Figure 7d shows the average transaction throughput. The x-axis is the number of membership levels per region in the analytics table. With a lower number of membership levels, there are fewer rows in the table, and the data contention is higher. Larger values in the x-axis indicate lower contention levels.

At high data contention with only one membership level per region, transactions take significantly longer to complete for 2PL and OCC. Specifically, 2PL experiences a 22% increase in latency, while OCC experiences a doubling of latency. OCC performs much worse than 2PL because it needs to abort and rerun transactions that fail the validation phase. For both IC and TC, the throughput and latency are not impacted by the change of data contention. This is because the ChopperGraph analysis guarantees serializable execution without locking resources across distributed hops (piece-wise execution), significantly reducing the contention between transactions.

E. *Scalability*

We evaluate the scalability of *ImmortalChopper* by increasing the number of edge areas. Each edge area contains one client and two servers. Each client sends 20 concurrent requests. With more edge areas, the number of total concurrent requests in the system will also increase, meaning the load on the system becomes higher.

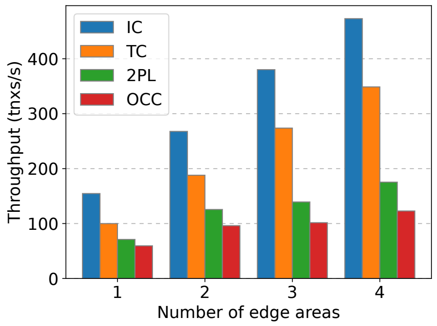
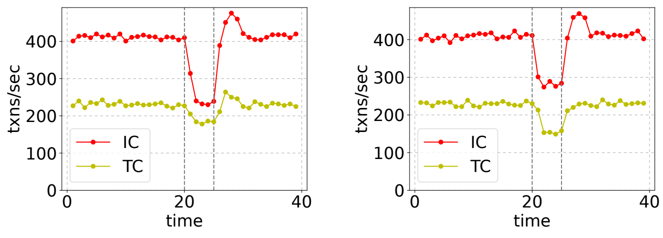


Fig. 8: Throughput with increasing number of areas



(a) Edge node failure (b) Network disconnection

Fig. 9: Throughput for failure and recovery

Figure 8 shows that IC scales well when increasing the number of areas. The throughput is increasing by 1.73x, 2.46x, and 3.06x for 2, 3, and 4 areas, respectively, compared to the single-area throughput. TC also shows similar scalability to IC. 2PL and OCC scale well from 1 area to 2 areas, but the throughput is not increased by much when there are 3 and 4 areas. The result shows that *ImmortalChopper* has better scalability than 2PL and OCC. This is because the piece-wise execution allows *ImmortalChopper* to process transactions with fewer messages between nodes and fewer operations in each node.

F. *Failure and Recovery*

We run an experiment to evaluate *ImmortalChopper*'s tolerance to two different failures, edge node failure and network disconnection. Edge node failure means the edge node cannot be accessed by any other node, while network disconnection means the edge node is disconnected from other edge nodes and backup nodes but the clients can still initiate transactions on the edge node. In the experiment, each client keeps 10 "alive" transactions concurrently — If a transaction is waiting for the edge node to recover, we put it to sleep and invoke a new transaction to keep the same number of "alive" transactions. The local transactions ratio is set to 50%.

TABLE I: Completion Latency of RUBIS and SOCIAL

Transactions	IC First	IC	TC	2PL	OOC
PutBid	1ms	94 ms	134ms	237ms	264ms
BeFriend	1ms	279 ms	319ms	319ms	320ms
PostStatus	1ms	94 ms	135ms	134ms	135ms

TABLE II: Throughput of RUBIS and SOCIAL

Transactions	IC	TC	2PL	OOC
PutBid	421	373	213	189
BeFriend	105	92	93	92
PostStatus	110	93	93	93

Figure 9 shows the throughput of the system along a timeline. The x-axis shows the experiment time, and the y-axis shows the throughput (sum of the four clients). An edge node experiences one of the above-mentioned failures at time 20, and it recovers after time 25.

Figure 9a shows the results of edge node failure. For IC, there is a throughput drop after the node failure at time 20. The reason is that transactions that access stale data cannot be committed until the edge node recovers. However, those transactions are already processed and are pending to be safely committed — the processing resources from the other edge nodes are not wasted. After time 25, the failed edge node recovers, and the system starts to resolve the Non-Safe hops. The throughput of the system quickly grows and has a higher throughput than the normal-case throughput. The reason is that after recovery, we can utilize all the work that was done by the backup node and the other edge nodes during the failure. TC has a smaller performance drop during the failure. This is because the backup node has the up-to-date data, so all transactions that involve the failed node can be executed by the backup node directly. However, IC has a higher throughput than TC at all times.

Figure 9b shows the result of a network disconnection. Compared to edge node failure, TC experiences a higher performance drop because the local transactions at the disconnected edge node need to wait for the replication response from the backup node, so they cannot be committed. Meanwhile, IC has a much lower throughput drop. This is because local transactions can be processed by the edge node itself, without waiting for replication.

The experiment results show that *ImmortalChopper* effectively minimizes the impact of node failure and network disconnection. It can quickly recover with the help of speculative execution.

G. Performance on Common Applications

We evaluate *ImmortalChopper* with two common applications: an auction service application (RUBIS) [26], and a social networking application (SOCIAL). We run each application with 8 nodes and 80 concurrent requests. The network latency configuration is the same as in the previous experiments. Due to the space limit, we show part of the results with *ImmortalChopper* only.

RUBIS is a web application built on a single-machine database where most of the operations are OLAP operations. We focus on the most important transaction of RUBIS, called $PutBid$. It contains two hops: the first hop inserts a new bid record (with the bidder's id and the bid price), the second hop updates the item's highest bid price if the new bid price is greater than the current highest bid price.

SOCIAL contains five tables ($Graph$, $User$, $Status$, $Wall$, and $Activity$). There are two important transactions: (1) $BeFriend$ transaction, where a user becomes friends with another user. It consists of 4 hops, two for inserting two edges into the $Graph$ table, and two for inserting $BeFriend$ logs into the $Activity$ table. (2) $PostStatus$ transaction, where a user updates their current status. It consists of 2 hops, one for updating the $Status$ table and one for inserting a $PostStatus$ activity. In the experiment, we run a mixed workload with an even distribution of these two transactions.

Table I and II show the performance of the two applications. *ImmortalChopper* achieves 1ms first-hop latency on all transactions, showing that the user can receive a response in real-time. $PutBid$ is a 2-hop transaction with high data contention. IC outperforms other baselines since it is not blocked by concurrent data competition. For $BeFriend$ and $PostStatus$, which are transactions without contention (all operations are insertions), IC outperforms 2PL and OCC by hiding the replication latency.

VI. RELATED WORKS

Edge-Cloud Systems. Edge-cloud systems allows data to be stored and processed close to the user. Many existing systems focus on improving the performance of a single edge node where most of the processing can be done locally [27]–[30]. For example, DuckDb [27] provides an embeddable database that can run on resource-limited edge nodes. Another popular direction in edge-cloud systems is to manage edge-node resources efficiently so user requests can be delegated to available nearby edge nodes [31]–[35]. There are also works that discuss how to coordinate data between edge-to-edge, or edge-to-cloud efficiently [4], [36]–[39]. Providing serializable distributed transactions in the edge-cloud environment is not well-studied in previous works due to the high coordination overhead between the far-away edge nodes. For example. TransEdge [40] uses traditional 2PC for distributed transaction coordination, which is much slower compared to our system.

Distributed Transaction Protocols. There are lots of distributed transaction protocols [11], [16], [41]–[43] designed for different applications with specific assumptions or varying guarantees. Some works focus on reducing the number of messages in the protocols. Primo [41] supports write-conflict-free concurrency control to avoid conflicts in the commit phase and thus reduces 2PC round-trips. Other works make use of the semantic knowledge, derived from the deterministic set of transactions, to remove unnecessary data conflict with a better execution plan [11], [16], [44]. Advanced hardware like RDMA is also utilized to accelerate distributed transactions [43], [45], [46]. To the best of our knowledge, no distributed transaction protocol can fully utilize the nature of data locality to achieve 1n-Response while tolerating node failures in the edge-cloud environment.

VII. CONCLUSION

In this paper, we present *ImmortalChopper*, a distributed transaction protocol designed for edge-cloud environments. It achieves 1n-Response by extending the transaction chopping theory to ChopperGraph, which integrates lazy replication of the first hops. The experiments demonstrate that *ImmortalChopper* provides 1n-Response in real-time and can tolerate edge node failure while providing efficient recovery.

VIII. AI-Generated Content Acknowledgement

There is no AI-Generated Content is this paper.

References

[1] A. R. Biswas and R. Giaffreda, "Iot and cloud convergence: Opportunities and challenges," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*. IEEE, 2014, pp. 375–376.

[2] M. Chiang and T. Zhang, "Fog and iot: An overview of research opportunities," *IEEE Internet of things journal*, vol. 3, no. 6, pp. 854–864, 2016.

[3] M. Merenda, C. Porcaro, and D. Iero, "Edge machine learning for ai-enabled iot devices: A review," *Sensors*, vol. 20, no. 9, p. 2533, 2020.

[4] D. Abadi, O. Arden, F. Nawab, and M. Shadmon, "Anylog: a grand unification of the internet of things," in *Conference on Innovative Data Systems Research (CIDR '20)*, 2020.

[5] J. Lorincz, L. Chiaraviglio, and F. Cuomo, "A measurement study of short-time cell outages in mobile cellular networks," *Computer communications*, vol. 79, pp. 92–102, 2016.

[6] S. Segan. (2022) No signal: The worst cellular dead zones in the us. [Online]. Available: https://www.pcmag.com/news/no-signal-the-worst-cellular-dead-zones-in-the-us

[7] P. A. Bernstein, V. Hadzilacos, N. Goodman *et al.*, *Concurrency control and recovery in database systems*. Addison-wesley Reading, 1987, vol. 370.

[8] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, vol. 30, no. 4, pp. 68–74, 1997.

[9] ——, "Fault-tolerance by replication in distributed systems," in *International conference on reliable software technologies*. Springer, 1996, pp. 38–57.

[10] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[11] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez, "Transaction chopping: Algorithms and performance studies," *ACM Transactions on Database Systems (TODS)*, vol. 20, no. 3, pp. 325–363, 1995.

[12] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 360–391, 1992.

[13] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li, "Transaction chains: achieving serializability with low latency in geo-distributed storage systems," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 276–291.

[14] D. Shasha, E. Simon, and P. Valduriez, "Simple rational guidance for chopping up transactions," in *Proceedings of the 1992 ACM SIGMOD International Conference on management of Data*, 1992, pp. 298–307.

[15] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, pp. 79–103, 2006.

[16] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD international conference on management of data*, 2012, pp. 1–12.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[18] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss *et al.*, "Cockroachdb: The resilient geo-distributed sql database," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 1493–1509.

[19] N. Bronson, A. Aghayev, A. Charapko, and T. Zhu, "Metastable failures in distributed systems," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 221–227.

[20] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko, "Metastable failures in the wild," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 73–90.

[21] C. D. Nguyen, K. Chen, C. DeCarolis, and D. J. Abadi, "Are database system researchers making correct assumptions about transaction workloads?" *Proceedings of the ACM on Management of Data*, vol. 3, no. 3, pp. 1–26, 2025.

[22] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[23] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[24] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, pp. 133–160, 2006.

[25] Chameleon. (2024) Chameleon cloud. [Online]. Available: https://www.chameleoncloud.org/

[26] Amza, Chanda, Cox, Elnikety, Gil, Rajamani, Zwaenepoel, Cecchet, and Marguerite, "Specification and implementation of dynamic web site benchmarks," in *2002 IEEE international workshop on workload characterization*. IEEE, 2002, pp. 3–13.

[27] M. Raasveldt and H. Mühleisen, "Duckdb: an embeddable analytical database," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1981–1984.

[28] J. Paparrizos, C. Liu, B. Barbarioli, J. Hwang, I. Edian, A. J. Elmore, M. J. Franklin, and S. Krishnan, "Vergedb: A database for iot analytics on edge devices." in *CIDR*, 2021.

[29] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "Riot: An open source operating system for low-end embedded devices in the iot," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[30] S. Venkataramani, K. Roy, and A. Raghunathan, "Efficient embedded learning for iot devices," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 308–311.

[31] H. Liu, F. Eldarrat, H. Alqahtani, A. Reznik, X. De Foy, and Y. Zhang, "Mobile edge cloud system: Architectures, challenges, and approaches," *IEEE Systems Journal*, vol. 12, no. 3, pp. 2495–2508, 2017.

[32] Y. Chen, J. Zhao, Y. Wu, J. Huang, and X. S. Shen, "Qoe-aware decentralized task offloading and resource allocation for end-edge-cloud systems: A game-theoretical approach," *IEEE Transactions on Mobile Computing*, 2022.

[33] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM transactions on networking*, vol. 24, no. 5, pp. 2795–2808, 2015.

[34] A. Ceselli, M. Premoli, and S. Secci, "Mobile edge cloud network design optimization," *IEEE/ACM Transactions on Networking*, vol. 25, no. 3, pp. 1818–1831, 2017.

[35] W. Zhang, J. Chen, Y. Zhang, and D. Raychaudhuri, "Towards efficient edge cloud augmentation for virtual reality mmogs," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–14.

[36] T. Guo, R. J. Walls, and S. S. Ogden, "Edgeserve: efficient deep learning model caching at the edge," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 313–315.

[37] S. Gazzaz, V. Chakraborty, and F. Nawab, "Croesus: Multi-stage processing and transactions for video-analytics in edge-cloud systems," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2022, pp. 1463–1476.

[38] D. O'Keeffe, T. Salonidis, and P. Pietzuch, "Frontier: Resilient edge processing for the internet of things," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1178–1191, 2018.

[39] A. M. Ghosh and K. Grolinger, "Edge-cloud computing for internet of things data analytics: Embedding intelligence in the edge with deep learning," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 3, pp. 2191–2200, 2020.

[40] A. A. Singh, A. Khan, S. Mehrotra, and F. Nawab, "Transedge: Supporting efficient read queries across untrusted edge nodes." in *EDBT*, 2023, pp. 684–696.

[41] Z. Lai, H. Fan, W. Zhou, Z. Ma, X. Peng, F. Li, and E. Lo, "Knock out 2pc with practicality intact: a high-performance and general distributed transaction protocol," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 2023, pp. 2317–2331.

[42] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: distributed transactions with consistency, availability, and performance," in *Proceedings of the 25th symposium on operating systems principles*, 2015, pp. 54–70.

[43] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 185–201.

[44] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting more concurrency from distributed transactions," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 479–494.

[45] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy, "Xenic: Smartnic-accelerated distributed transactions," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 740–755.

[46] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, "Fast and general distributed transactions using rdma and htm," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–17.