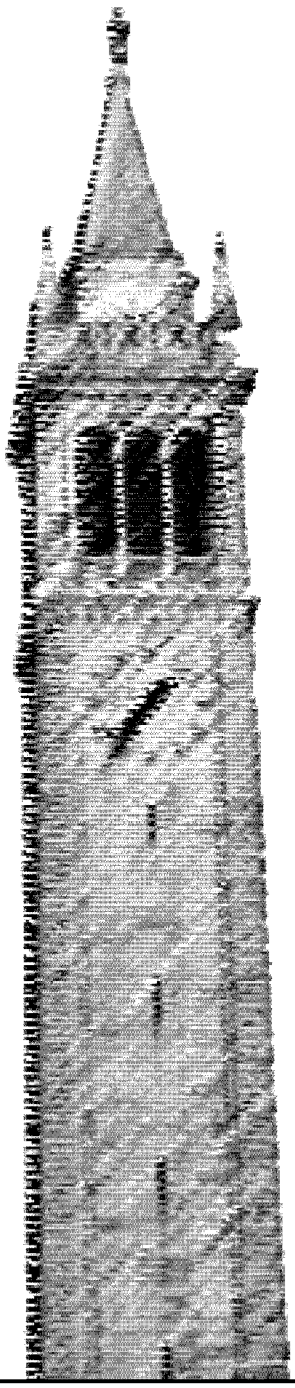
Fault Tolerance in Distributed Systems



Aisha Mushtaq

Fault Tolerance in Distributed Systems

by

Aisha Mushtaq

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Spring 2022

Fault Tolerance in Distributed Systems

Abstract

Fault Tolerance in Distributed Systems

by

Aisha Mushtaq

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Co-chair

Professor Sylvia Ratnasamy, Co-chair

Modern datacenter applications need to provide resiliency to mask failures. These applications widely use techniques like Replicated State Machines to provide fault tolerance. Replicated State Machines typically rely on consensus protocols to provide availability and consistency. These applications also require high throughput and low latency from the underlying consensus protocols. Furthermore, in an effort to further reduce latency experienced by clients, we are seeing the emergence of edge computing; storage and computational resources are placed in between the clients and servers in datacenters (typically closer to the client). This placement provides many benefits: lower-latency responses to clients, lower bandwidth demands on the backbone and increased privacy. Stateful applications running on the edge pose a problem of losing state when an edge node fails.

This dissertation looks at fault tolerance for datacenters and edge computing. First, RingWorld looks at datacenter fault tolerance for highly available lock services. RingWorld adapts ring-based consensus protocols to leverage programmable switches and datacenter topology. This allows RingWorld to provide higher throughput and comparable latency to existing lock services. Second, to provide fault tolerance for edge computing, CESSNA provides a mechanism to recover from edge failures for strongly stateful applications that ensures correctness and good performance. To do this, CESSNA defines the consistency guarantee for correctness in the face of edge failures, and recovers from failures by adapting techniques like log-replay.

To mom, Asad and Azoo

Contents

Acknowledgments

I am immensely grateful to have been a part of the NetSys lab at Berkeley. I made many great friends and learned from great teachers. This dissertation would not have been possible without these amazing people!

Scott Shenker. The wise ol' Scott has been my North Star throughout my PhD process and has guided me through many ups and downs (many downs!). Whenever I was feeling lost and hopeless, Scott would meet me brimming with positivity and would share lots of ideas, and eventually, we'd figure out a way forward. He's always treated me with lots of patience and kindness and has always had anecdotes to share through any problems I'd face, be they personal or academic.

Sylvia Ratnasamy. Sylvia with her candid feedback has always encouraged me to reach my potential. I have not spent as much time with her as I would have liked, but whenever I have approached her, she has always provided thoughtfulness and clarity and has helped me critically examine the deeper parts of my research.

Aurojit Panda. I started collaborating with Panda later in my PhD career, and I regret it often. Panda's immense knowledge of anything and everything is unbelievable. Over the course of my projects, he has guided me through all things great and small. I've bugged him many times with insignificant things; he's always quick to respond, but patient as well in explaining stuff again and again.

Yotam Harchol. Yotam has been my mentor and my friend. I watched him do research and I was so impressed, I've been trying to emulate him ever since. He has taught me how to look at problems, how to run experiments, document them, how to write and so much more. His mentorship has helped me become a better researcher.

James 'Murphy' McCauley. I enjoyed our conversations together. Murphy and I could talk for hours on end and I would always come out learning something new from him. He could listen to my nonsensical ramblings and then translate them into something useful. Thank you for humoring me.

Jon Kuroda. Without Jon's help, I probably would not have been able to graduate. I approached Jon to help set up the infrastructure for RingWorld. The setup required setting up four programmable switches and infinite connections with a two-day deadline! He magically set it up with such elegance and thoroughness. Thank you Jon!

Natacha Crooks and Dan Ports. My collaborators on RingWorld provided me with valuable insights that helped me polish my research. I learned a great deal from both of them.

Amin Tootoonchian, Amy Ousterhout, Anwar Hithnawi, Christopher-Branner Augmon, Ed Oakes, Emmanuel Amaro, Ethan Jackson, Lloyd Brown, Michael Chang, Narek Galstyan, Radhika Mittal, Sarah McClure, Silvery Fu, Vivian Fang, Wen Zhang, and Zhihong Luo. The time I spent with friends and collaborators during my time at the NetSys Lab has been invaluable. I enjoyed our lunch conversations, laughter, and chocolates.

To my family, and my friends in the Bay Area, for their continuous support and understanding throughout my PhD.
To my husband **Asad**. I owe my Phd to you; if it were not for your unwavering support and help I would not be writing this dissertation. You've been with me through it all, the ups, the downs, and the tears. Whenever I felt I could not do it anymore, you would jump in, you'd discuss ideas, run experiments, and more, without ever asking for anything in return. I am really glad to have you by my side.
To my baby **Azlan**, you were the calmest baby ever, I don't know how I would have done it all otherwise. You have the warmest smile that will melt anyone's heart!
To my **mom**, thank you for always being there for me and for all your prayers that helped me.

Chapter 1

Introduction

Datacenters power many of today's services. These services run on clusters comprised of hundreds or even thousands of commodity servers. Failures in such large clusters of servers are inevitable. The multiplicative effect of individual failure rates - compared to that of a single server - means failures are expected every few hours or less [5]. However, despite the unreliability of the underlying infrastructure, applications need to work around failures to provide uninterrupted service.

To guarantee that services remain available, today's applications relies on fault-tolerance techniques like replicated state machines (RSMs). Replication stores state on multiple replicas to ensure key services remain available and consistent. For example, Chubby [10], Etcd [21], and ZooKeeper [30] use replication to build highly available lock services that are widely used to coordinate access to shared resources and configuration information. Similarly, persistent storage systems, such as Spanner [12] and H-Store [64], use replication to prevent system outages or data loss. Replication typically rely on consensus protocols such as Paxos [67], Viewstamped Replication [51, 43], and Raft [52] to keep replicas consistent. Modern applications simultaneously require high throughput and low latency from their underlying consensus protocols. However, many factors, *e.g.,* communication overheads, limit the throughput of these protocols. Additionally, the performance of these protocols scales inversely with the number of replicas; *i.e.,* lower throughput and higher latency as the number of replicas increase.

It is becoming increasingly important for modern applications to keep total latency within strict bounds. To this effect, we are seeing the emergence of edge computing with computational and storage resources being deployed at the network edge. This allows to offload computation/storage from clients and/or servers (in datacenters). These edge servers are also prone to failures like servers in datacenters. While solutions for datacenter fault tolerance, such as RSMs, are well studied in the literature, these solutions do not directly apply to edge fault tolerance. For instance, RSMs are not a suitable choice for edge fault tolerance because (a) edge locations have limited resources (it is not possible to scale thousands of edge locations) and (b) entire edge sites can fail (it is too expensive to provide adequate compute and power redundancy to all edge sites). One might suggest far-site replication to provide resiliency, however, this can be too slow and wash out the latency benefits of the edge.

In this dissertation, we explore fault tolerance from two perspectives: (i) datacenters and (ii) edges. To that effect, we explore the answers to the following questions: (i) How to achieve consensus in datacenters with high throughput and comparable latency to existing RSM protocols? and (ii) How to provide resiliency for edge computing based on the constraints of edges?

We address the first question by presenting RingWorld, a new protocol for achieving consensus in datacenters. We then answer the second question using CESSNA, a protocol to provide resiliency for edge applications. We elaborate on these two protocols next.

1.1 RingWorld: Datacenter Fault Tolerance

RingWorld is a new consensus protocol for datacenters that leverages a ring-of-racks approach; each rack contains a programmable top-of-rack (ToR) switch and multiple servers. Ring-based consensus protocols arrange the replicas in a logical ring. This structure allows them to achieve optimal throughput regardless of the number of replicas. However, the latency of these protocols grows linearly as we increase the number of replicas. In RingWorld, the processing is split between a fast-path ToR ring and a slow-path on the server. This approach allows RingWorld to achieve the latency of the commonly used consensus protocols while providing the throughput benefits of ring-based protocols as the number of replicas scale.

1.2 CESSNA: Edge Fault Tolerance

The introduction of computational resources at the network edge allows application designers to offload computation from clients and/or servers, thereby reducing response latency and backbone bandwidth. More fundamentally, edge-computing moves applications from a client-server model to a client-edge-server model. While this is an attractive paradigm for many use cases, it raises the question of how to design client-edge-server systems so they can tolerate edge failures and client mobility. This is particularly challenging when edge processing is strongly stateful. In chapter 3, we propose a design for meeting this challenge called the Client-Edge-Server for Stateful Network Applications (CESSNA).

1.3 Co-Authored/Previously Published Work

In this dissertation, the work for chapter 2 was done in collaboration with Aurojit Panda, Dan Ports, Natacha Crooks, and Scott Shenker. The work for chapter 3 was done in collaboration with Yotam Harchol, Vivian Fang, James McCauley, Aurojit Panda, and Scott Shenker. The material in chapter 3 is an adaptation from [28].

Chapter 2

Fault Tolerance in Datacenters

Highly available lock services such as Chubby [10], Etcd [21], ZooKeeper [30], and other similar services are a core component of many distributed applications deployed in modern datacenters. Many applications, including distributed databases that provide data replication, and cluster management systems, rely on these services to implement mutual exclusion between processes, to track resource allocation, and for coordination. These services must be fault tolerant, and are usually implemented as replicated-state machines (RSMs) using consensus protocols such as Paxos [67] (Chubby), Raft [52] (Etcd), and ZAB [34] (ZooKeeper).

Many factors, including the protocol they use and RPC and communication overheads, limit the throughput of existing services. While using more replicas increases fault tolerance, consensus protocol performance scales inversely with the number of replicas; *i.e.,* these protocols exhibit lower throughput and higher latency as the number of replicas increases. Consequently, application developers generally try to minimize the use of these lock services [10]. This avoidance often requires changing application semantics; *e.g.,* requiring that applications weaken consistency guarantees to avoid the need for coordination. In cases where an application cannot avoid coordination, the lock service often becomes a scaling bottleneck. Thus, the limited throughput of current lock services imposes limits on both application semantics and performance.

In this chapter, we look at the question of how to build a consensus protocol for locking services that provides higher throughput than current solutions while still having latency that is comparable to existing lock services. Our starting point is ring-based consensus protocols such as Carousel [25] and Ring Paxos [45]. Ring-based protocols arrange replicas in a logical ring, and in most cases, a replica only receives messages from its predecessor and only sends messages to its successor. This structure means that the number of messages sent or received by a replica is independent of the number of replicas deployed. Additionally, this communication structure also makes it easier to pipeline proposals (or requests) and thus allows throughput to scale as the number of replicas increases. Indeed, prior work [25] has shown that ring-based protocols are throughput optimal. Unfortunately, these throughput improvements come at the cost of latency that grows linearly as the number of replicas (and hence ring size) increases.

In this chapter, we propose RingWorld, a new protocol that leverages programmable switches to modify the ring-based approach so that it can match the latency of more commonly used broadcast-based protocols (*e.g.,* Raft, Paxos, and ZAB) while providing the throughput benefits of ring-based protocols. Instead of a ring of servers, RingWorld runs on a ring of *racks*, each of which contains a programmable top-of-rack switch (ToR) and multiple servers. Processing is split between a fast-path running on the ToR switches and a slow-path running on the server. The switch fast-path is responsible for maintaining consistent ordering between requests, for forwarding messages to the successor rack, and for broadcasting messages and collecting responses from all servers in the rack. Servers are responsible for actually replicating and maintaining the state machine. Because the ring propagation time is small compared to the server processing time, the response latency (in contrast to typical ring-based protocols) does not grow with the size of the ring (until the ring reaches very large sizes, far beyond what is needed or desired).

While RingWorld's design minimizes latency increase due to message transfer between racks, the actual latency for the consensus protocol can still be high due to straggling servers[1] (a sufficient number of which must acknowledge that they have replicated state) or because of failures. RingWorld handles straggling servers by having ToRs temporarily disconnect servers (*i.e.,* limit the packet traffic to/from them) who are late in responding. RingWorld combines this with a protocol for server recovery that, in most cases, only involves its associated ToR. Similarly, the ring-based nature of RingWorld allows ToR failures to be handled locally (by the successor ToR) without requiring a more expensive view change operation. As we show later in §2.4, these choices allow RingWorld to provide similar response latencies as existing lock services while providing better throughput. We have implemented and tested RingWorld on a cluster of four Tofino based programmable switches and twelve servers. We describe the protocol and our implementation in greater detail in later sections of the chapter.

We are of course not the first to propose using programmable switches to improve lock service performance and others, notably NetLock [71] have also used programmable switches to improve lock-service performance. However, these prior efforts focused on reducing latency and hence focused on optimizing broadcast based consensus protocols. Furthermore, these prior efforts moved all state to switches, which in turn required them to both limit the amount of state they could support and the types of operations they could implement. RingWorld's design limits neither. We delay a detailed discussion of the related work to §2.5.

Next, we present some background on programmable switches and consensus protocols before presenting RingWorld's design.

[1]Programmable switch ASICs are designed to process all packets in a constant number of clock cycles, and hence switches cannot be stragglers in our setting.

2.1 Background

2.1.1 Safety and Liveness for RSMs

As noted before, replicated state machines (RSMs) [60] are a common abstraction used for building fault-tolerant distributed services. RSMs are generally implemented using consensus protocols that are used to build a consistent log replicated across several processes. This consistent replicated log ensures that processes can fail without impacting application correctness (safety). These protocols also ensure that the service remains available as long as a certain number of processes remain functioning (liveness).

Consensus protocols have been widely studied in the distributed systems literature, and a correct consensus protocol should ensure three safety properties [22] (which are most easily stated in the binary consensus framework [22]):

- **Agreement:** All correct processes must agree on the same value, for RSMs this means all processes must agree on the contents of the log.
- **Validity:** The value at any correct process must be the input to some correct process.
- **Termination:** All processes must eventually decide on a value, *i.e.,* all correct processes must eventually reach an agreement.

The well-known FLP result [22] shows that no fault-tolerant protocol designed to run on an asynchronous network can ensure all three properties. As a result, protocols, such as Paxos [38], ensure agreement and validity, but cannot ensure termination.

2.1.2 Programmable Switches

Programmable switches, such as Intel Tofino [31] and Broadcom Trident [9], enable the ability to add functionality beyond plain packet forwarding. These switches provide flexible packet parsing and a programmable match-action pipeline to enable programmability. Packet header and metadata information is stored in the packet header vector (PHV) after parsing. The ingress and egress match-action pipelines have multiple stages. Objects are allocated in each stage and accessed by the packets via ALUs. The switches use a limited amount of on-chip memory to provide stateful elements which include tables, registers, and counters. Additionally, these switches provide other built-in functionality such as packet resubmission, recirculation, and mirroring for more advanced packet processing.

Programmable switches have high throughput and deterministic packet processing latencies. However, they achieve their performance by restricting program semantics. These switches have a limited number of stages, with each stage only allowing a fixed amount of computation. They are more suited for a flat, linear programming model with limited branching. They support reduced comparison capabilities, such as equality comparisons

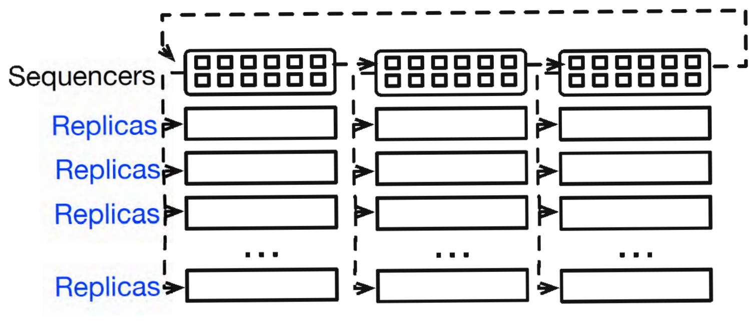


Figure 2.1: RingWorld topology.

between variable fields (limited in bitwidth) and less than/greater than comparison with constants only.

Additionally, these switches have access to a limited amount of state that can only be accessed in a specific order (*i.e.,* the stateful element is tied to a particular stage). Amongst the stateful elements, only registers provide both read and write capabilities in the dataplane, while tables are read-only, and counters are write-only. For registers, all read-modify-write operations are atomic. We can not write arbitrary values to the registers; the value must either be a constant, a modification of the previous value, or a value from the packet header vector (header/metadata). The registers only allow a single comparison with values from the packet header vector. A register array indicates a register with n entries. We can not access a register array at two different indices in a single pass of the pipeline.

2.2 System Model

A RingWorld deployment consists of at least $2r + 1$ racks, each of which contains at least $2f+1$ servers and one top-of-rack switch. We assume fail-stop failures for both ToRs (which, in our model, encompasses all rack-wide failures) and servers. Further, we assume no more than r ToRs can fail and no more than f servers in a rack can fail.

In terms of communication, we assume that any pair of ToRs can reach each other through the datacenter network. In addition, we assume that the network is asynchronous but reliable, *i.e.,* messages can be arbitrarily delayed but may not be dropped or reordered. This does not require a reliable physical network, but can be achieved through the use of reliable tunnels (*e.g.,* GRE tunnels) or by enabling priority flow control (which is often used when deploying RDMA).

Logically, the ToRs (and their associated racks) are arranged in a ring (as shown in figure 2.1), with each ToR assigned a unique ID based on its relative location in the ring. Each ToR knows the entire mapping from IDs to ToRs. Similarly, we assign IDs to each server. IDs must be unique within a rack but need not be unique across racks, and each ToR can

State	Type	Description
`tid`	Integer	ToR ID.
`next`	Integer	Successor's ToR ID.
`prev`	Integer	Precessor's ToR ID.
`f`, `r`	Integers	Failure parameters.
`prop`	Integer	Proposal ID, initially 0.
`last_prop`	Integer	Highest Proposal ID seen so far. Previous sequence tracking.
`last_prop_tid`	Register array: proposal id to ToR id	Previous sequence tracking.
`acks`	Register array: seq no.-server bool bitmap	Acknowledgement tracking.
`acks_sum`	Register array: seq no.-ack count	Acknowledgement tracking.
`children`	Set of Integers	Correct servers in the rack.

Table 2.1: Configuration and state at each ToR.

State	Type	Description
`id`	Integer	Server ID,
`parent`	Integer	ToR ID to which server is connected.
`pending`	Priority queue of messages	Messages pending delivery.
`can_deliver`	Dictionary: seq no. to bool	Can the packet be delivered.

Table 2.2: Configuration and state at each server.

map machine IDs to ports. During normal operations, ToRs only send protocol messages to directly connected servers or to their immediate successor.

Similar to other protocols RingWorld guarantees safety and liveness when no more than r ToRs fail and no more than f servers in a rack fail. For liveness, we additionally expect sufficient periods of synchrony to make progress.

2.3 RingWorld

In this section, we describe the design of RingWorld. Similar to all RSM protocols, RingWorld primarily provides a mechanism to append values to an ordered log which is replicated

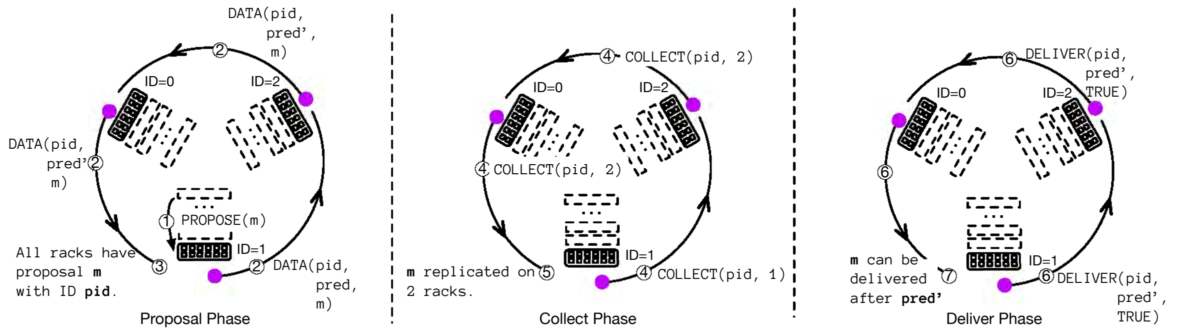


Figure 2.2: An overview of the RingWorld protocol in the absence of failures.

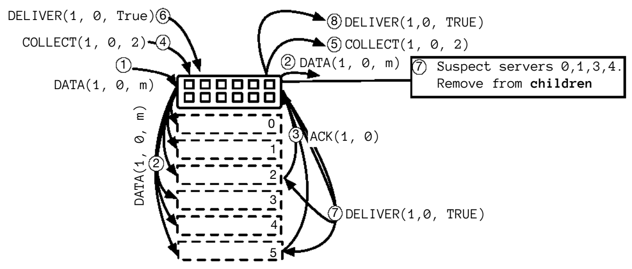


Figure 2.3: A look at messages within rack 0 in the execution shown in Figure 2.2.

across servers in a consistent manner. RingWorld also provides mechanisms to recover from server and rack failures and to change the set of participants.

RingWorld replicates state across racks. Within a rack, RingWorld's logic and state are partitioned across the top-of-rack(ToR) switch and servers. Due to the limited processing capabilities of switch ASICs and the limited amount of on-board memory, RingWorld is designed to minimize the amount of state stored at the switch and accessed when processing messages at the ToR. As a result, servers are responsible for maintaining the actual contents of the replicated log. We list the state stored at a rack's ToR in Table 2.1 and at servers in Table 2.2. We explain the semantics and contents of each state element as we walk through the algorithm.

Servers in RingWorld are also responsible for receiving requests from clients and responding to these requests. RingWorld is a leaderless protocol, and as a result, any server located in any participating rack can process client requests. We do not make any assumptions about how client requests are distributed across servers, and instead assume that some external mechanism (*e.g.,* a load balancer) is responsible for this task. For ease of exposition, we assume that each request must be successfully appended to the log (*i.e.,* committed) before a response is sent back to the client.

2.3.1 Overview

Before delving into the details of the RingWorld protocol, we first describe a brief overview of the protocol.

Initiating proposals: Upon receiving a client request, a server sends a `propose` message to the ToR in its rack, which then assigns a unique sequence number and a previous sequence number to the proposal to produce a `DATA` message. The committed proposals in RingWorld's totally-ordered log is ordered according to the lexicographic order of these sequence numbers. The proposing ToR then initiates the process of appending an entry to the log which proceeds in three phases (shown in figure 2.2 and figure 2.3).

Phase 1 - Proposal: In the proposal phase, the proposal is replicated across the ring. The ToR broadcasts the `DATA` message to all the servers in its rack and concurrently forwards the message to its successor ToR in the ring, which in turn performs the same actions. Upon receiving the proposal, a server adds the proposal to its temporary log (pending) in the lexicographic order of the sequence number and sends an acknowledgment to the ToR. The first phase ends when the message completes a round around the logical ring, *i.e.,* when the proposing ToR receives the `DATA` message back.

Phase 2 - Collect: The proposing ToR triggers the next phase by sending a `COLLECT` message around the ring. This phase collects the following piece of information: whether the proposal has been sufficiently replicated across the racks in the ring. We consider a proposal to be sufficiently replicated if and only if it has been replicated by at least $f + 1$ servers. ToRs maintain a count of how many servers in their rack have acknowledged a proposal and mark in the `COLLECT` message if a proposal has been sufficiently replicated (or not) by incrementing (or leaving unchanged) the count of racks that have replicated the proposal. The collect phase ends when the message completes a round around the ring.

Phase 3 - Deliver: Finally, the proposing ToR uses the count from the `COLLECT` message to determine whether the proposal can be committed or not. Once determined, the proposing switch disseminates this decision in the deliver phase by sending a `DELIVER` message which contains the decision. The message is forwarded around the ring, and each ToR broadcasts it to all servers in its rack. Upon receiving the `DELIVER` message, the servers mark the proposal to be committed or aborted. The servers move proposals from their temporary log to their consistent log if there are no other proposals before it that need to be committed/aborted.

2.3.2 Protocol

We now describe the RingWorld protocol in detail.

Ordering requests

Upon receiving a client request, a server sends a `propose` message to the ToR in its rack. This initiates the process of appending the message to the log. The rack's ToR assigns the proposal a sequence number of the form (`ToR ID`, `proposal ID`, where the `proposal ID` is a monotonically increasing integer that meets the following properties: i) The `proposal ID` assigned to a new proposal is larger than that assigned to any previously committed proposal regardless of the origin rack of those committed proposals and ii) The `proposal ID` is distinct from any other proposal sent by the ToR.

The committed proposals are ordered according to the lexicographic order of these sequence numbers. The constraints on `proposal ID` ensure that this is a total order and that the total order of proposals is a superset of the committed order, *i.e.,* this ensures that RingWorld maintains linearizability.

Additionally, each proposal is assigned a previous sequence number of the form (`ToR ID`, `proposal ID`). The previous sequence number indicates the sequence number that ought to be ordered before our current proposal in the consistent log. As we will see later, this prevents RingWorld from committing entries out of order in case of packet drops, thereby maintaining linearizability. The ToR initially assigns its highest known sequence number as the previous sequence number to the proposal which is updated as the proposal travels around the ring.

Packet headers

RingWorld adds a RingWorld header on top of the Ethernet header. The RingWorld header contains the following fields: `packet type`, `tor id`, `proposal id`, `previous proposal tor id`, and `previous proposal id`. Additionally, `collect` packets contain an `ack_count` field, and `deliver` packets contain a `decision` field. `ACK` packets also contain a `server id` field.

Initiating proposals

Listing 2.1 shows the pseudo-code for initiating a proposal at the ToR. The ToR assigns the proposal a sequence number and a previous sequence number and updates its own state. The state updates must be atomic (read-modify-write), so the ToR reads the register values and stores them temporarily in packet metadata and then updates the register values.

Additionally, the `last_prop_tid` register array is accessed at two different indices for a `propose` packet. This is not a permissible operation in current P4 switches. To get around this limitation, we need to resubmit `propose` packets at the ToR. The resubmit operation allows the switch to resubmit the packet to the beginning of the packet processing pipeline, so that the packet can go through and be processed by the pipeline again. Since we are resubmitting packets and reading/writing `last_prop_tid` in two rounds, we must also ensure consistent reads/writes. For this, we leverage three properties of the switch and the RingWorld protocol:

- All resubmitted `propose` packets arrive back in the same relative order at the ToR (though they might be interleaved with several other packets),
- Writes in `last_prop_tid` are done at monotonically increasing indices with each index being written to only once, and

```
procedure broadcast(message) {
    send(next, message)
    send(children, message)
}

callback receive PROPOSE(m) from server c {
  % Assign proposal a sequence number (tid, prop)
  % Assign proposal the previous sequence number (last_prop_tid[last_prop], last_prop)
  % Send proposal to all servers in the rack and successor.

  if resubmit == 0 {
    m_prop = prop % Read and assign proposal ID
    prop += 1 % Increment proposal ID

    prev_prop = last_prop % Read and assign prev proposal ID
    last_prop = m_prop % Update prev proposal ID to current m_prop i.e., highest
    proposal id seen so far

    last_prop_tid[last_prop] = tid % Write ToR ID corresponding to prev proposal ID (
    highest proposal ID seen so far)

    % Resubmit packet and propagate values of m_prop and prev_prop
    resubmit(m_prop, prev_prop, m)
  } else {
    prev_tid = last_prop_tid[prev_prop] % Read the value of last_prop_tid at prev_prop
    broadcast(DATA(tid, m_prop, prev_tid, prev_prop m))
  }
}
```

Listing 2.1: ToR logic for initiating a proposal in RingWorld.

- Reads are always performed on lower indices than the writes (which will not be updated again).

This ensures we always have consistent reads/writes across `propose` and, as we will see later, `data` packet resubmissions. Current P4 switches also do not allow modifying packet headers in a resubmission's first round, so we also need to propagate the values of `m_prop` and `prev_prop` using a resubmission header to ensure their consistency.

The ToR broadcasts the transformed `data` message to all servers in its rack and the next hop in the ring starting the first phase of RingWorld.

Phase 1 - Proposal

The aim of the proposal phase is to replicate the proposal across racks. Listings 2.2-2.4 and 2.5 show the pseudo-code for RingWorld's proposal phase for the ToR and server respectively.

In the proposal phase, the `data` packet is circulated around the ring of ToRs, which in turn broadcast the packet to all servers in their rack for replication.

Upon receiving a `data` packet that was not proposed by the ToR itself, the ToR updates the previous sequence number information in the packet header and updates its local state. Similar to the `propose` packets, `data` packets need to access the `last_prop_id` register array at two indices, which requires a packet resubmission. Additionally, `data` packets also require several less-than/ greater-than comparisons to update the `prev_prop` field in the packet header. These comparisons are not directly supported in current P4 switches. To get around this limitation, we check the most significant bit of the difference between the two values to be compared. Occasionally, we also need to split the comparisons into smaller bit length comparisons. Finally, we split the computation between ingress (line 3) and egress (line 32) pipelines to fit the computation in the limited number of switch stages.

The packet is finally forwarded to all the servers in the rack and the next hop in the ring.

The servers upon receiving the `data` packet append the proposal in their `pending` log in the lexicographic order described above in section 2.3.2 and mark it as not deliverable yet. The servers also start a timer for the proposal. This is needed to prevent stalling of `pending` log processing in the case of ToR failure which we will describe later. The servers then respond with an `ACK` packet to the ToR.

Upon receiving an `ACK` packet from a server in its rack, the ToR updates the `acks` and `acks_sum` state. The `acks` register array tracks which servers have responded to a proposal, serving a two-fold purpose: (i) helps prevent double-counting of `ACKs` and (ii) checks which servers to take offline in the case of no response (described in phase 3). Due to limited stages in the switch, and the limited computation available to each stage, it is not possible to count the number of `ACKs` by checking the servers from `acks` register array. Therefore, to count the servers which have responded, we use the sum kept in `acks_sum` array.

Once a `data` packet completes a full round around the ring and arrives at the proposing ToR, the `data` packet will have the correct previous sequence number in its header. Upon receiving the `data` packet that was proposed by the ToR itself, the ToR transforms the packet into a `collect` packet and begins the second phase of RingWorld.

Phase 2 - Collect

The aim of the collect phase is to determine if the proposal has been sufficiently replicated across racks. A proposal is considered sufficiently replicated in a rack if and only if $\geq f+1$ servers have replicated the proposal. This ensures at least one live server in the rack always has the proposal. A proposal is considered committable if and only if $\geq r+1$ racks have sufficiently replicated the proposal.

Listing 2.6 shows the pseudocode for RingWorld's collect phase. The proposing ToR checks how many servers have responded with an `ACK` for the proposal and increments the `ack_count` in the packet header if the proposal has been sufficiently replicated. The `collect` then loops around the ring, collecting the same information from each ToR in the ring and updating the header.

```
callback receive DATA(t, m_prop, prev_tid, prev_prop, m) {
  if (t != tid) { % Received a proposal from a different rack.
    % Ingress processing
    if (resubmit == 0) {
      % Update the proposal count to ensure that all subsequent proposals from this
    rack will be ordered after.
      prop = max(prop, m_prop + 1)

      prev_prop_temp = last_prop
      last_prop = max(last_prop, m_prop)

      % Update last_prop_tid if the last_prop was updated i.e., if (m_prop >
    prev_prop_temp)
      % msb returns the most significant bit of the input.
      if msb(prev_prop_temp - m_prop) == 1 {
       last_prop_tid[last_prop] = t
      }
      resubmit(m_prop, prev_prop_temp, m)
    } else {
      % If this proposal is the highest sequence seen so far, set the index to read
    for last_prop_tid to prev_prop_temp (old highest sequence seen so far)
      if msb(prev_prop_temp - m_prop) == 1 {
        prev_prop_idx = prev_prop_temp
      } else { % If this proposal isn't the highest sequence seen so far, set the
    index to read for last_prop_tid to current proposal's prop id
        prev_prop_idx = m_prop
        % If the proposal's tid is greater than current tid, the highest seen proposal
     id for this proposal must be one less than the proposal's prop_id.
        if (msb(t - tid) == 1) {
          prev_prop_idx = prev_prop_idx - 1
        }
      }
      prev_tid_tmp = last_prop_tid[prev_prop_idx]
    }
    send_egress(next, DATA(tid, m_prop, prev_tid, prev_prop, m))
    send_egress(children, DATA(tid, m_prop, prev_tid, prev_prop, m))
  } else {
    % This is a proposal sent by this rack, and its receipt implies that all other
    racks have received the proposal.
    % Start the COLLECT phase.
    send_collect(tid, m_prop, prev_tid, prev_prop, 0)
}
```

Listing 2.2: ToR logic for RingWorld's proposal phase at the ingress pipeline.

```
callback receive DATA from ingress {
% If the ToR knows of higher sequence no. than the packet's previous sequence no.
% but less than current sequence no. update the packet's previous sequence no
  if ((msb(prev_prop - prev_prop_idx) == 1) or (prev_prop == prev_prop_idx
  and msb(prev_tid - prev_tid_tmp) == 1)) and ((msb(prev_prop_idx - m_prop) == 1)
  or (prev_prop_idx == m_prop and msb(prev_tid_tmp - t) == 1)) {
        prev_prop = prev_prop_idx
        prev_tid = prev_tid_tmp
  }
}
```

Listing 2.3: ToR logic for RingWorld's proposal phase at the egress pipeline.

```
callback receive ACK(t, m_prop) from server c {
  % Record the fact that server c has received an ack.
  if (c in children) {
    ack_count = acks[(t, m_prop)] % Read old value of acks
    acks[(t, m_prop)] = acks[(t, m_prop)] || c % Record an ack from server c, where c
    is the one-hot encoded server id.
    if (ack_count & c == 0) { % de-duplicate ACK, increment only if old value has not
    accounted for this server.
      acks_sum[(t, m_prop)] += 1 % Update the sum of acks for proposal
    }
  }
}
```

Listing 2.4: ToR logic for processing ACKs in RingWorld.

```
callback receive DATA(t, m_prop, prev_tid, prev_prop, m) {
  % insert DATA in sorted order of (m_prop, t)
  pending.insert((t, m_prop), m)
  can_deliver[t, m_prop] = null
  timer.start(t1)
  send(parent, ACK(t, m_prop))
}
```

Listing 2.5: Server logic for RingWorld's proposal phase.

```
% Check if a proposal is sufficiently replicated and then
% forward the appropriate collect message.
procedure send_collect(t, m_prop, prev_tid, prev_prop, count) {
    if (acks_sum(t, m_prop)]) > f) {
        % We consider a proposal to have been replicated on this rack
        % if and only if more than f servers have acknowledged the proposal.
        send(next, COLLECT(id, m_prop, prev_tid, prev_prop, count + 1))
    } else {
        send(next, COLLECT(id, m_prop, prev_tid, prev_prop, count))
    }
}

callback receive COLLECT(t, m_prop, prev_tid, prev_prop, ack_count) from prev {
  if (t != tid) { % A COLLECT message for a proposal from another rack.
  % Update count and send it to the successor.
    send_collect(t, m_prop, prev_tid, prev_prop, ack_count)
  } else { % A collect message originally sent by this ToR.
    % Count the number of racks that have successfully replicated this message to
    decide whether or not to commit.
    if (ack_count > r) { % Commit message (tid, m_prop)
      broadcast(DELIVER(t, m_prop, prev_tid, prev_prop, true))
    } else { % Cannot commit, mark.
      broadcast(DELIVER(id, m_prop, prev_tid, prev_prop, false))
    }
    % Update children to remove any who have not acknowledged the proposal until now.
    suspect_servers(id, seq)
  }
}
```

Listing 2.6: ToR logic for RingWorld's collect phase.

Since ToR-to-ToR latency is often lower than ToR-to-server latency, it is possible that the servers did not have enough time to respond with `ACKs` for the proposal by the time the collect phase starts. In that case, we can extend the collect phase's duration by looping the `collect` packet around multiple times. We can set the number of times we loop to balance the latency of the system and prevent excessive suspicion of servers having failed.

Once a `collect` packet arrives back at the proposing ToR, the proposing ToR determines if $\geq r+1$ racks have sufficiently replicated the proposal using the `ack_count` header. The proposing ToR then transforms the packet into a `deliver` packet and includes the decision to commit or abort the proposal. The `deliver` packet is broadcasted to the next hop in the ring and all servers in the rack which starts the third phase of RingWorld.

Phase 3 - Deliver

The deliver phase disseminates the decision (commit/abort) for the proposal around the ring. Listings 2.7 and 2.8 show the pseudocode for RingWorld's deliver phase for the ToR

```
procedure suspect_servers(t, seq) {
  % Servers that did not respond by deliver are removed from children.
  children = children && acks[(t,seq)]
}

callback receive DELIVER(t, m_prop, prev_tid, prev_prop, commit) {
  if (t != id) {
    broadcast(DELIVER(t, m_prop, prev_tid, prev_prop, commit)
    suspect_servers(t, seq)
  }
}
```

Listing 2.7: ToR logic for RingWorld's deliver phase.

```
procedure try_deliver() {
  % process messages from head only: ensures every machine would deliver in the same
    order
  while ((t, m_prop, m) = pending.remove()) {
     if (can_deliver[(t, m_prop)] && can_deliver[(link[t, m_prop])]) {
       deliver(m)
     } else {
        pending.insert((t, m_prop), m)
     }
  }
}

callback receive DELIVER(t, m_prop, prev_tid, prev_prop, commit) {
  cancel_timer(t1)
  can_deliver[t, m_prop] = commit
  link[t, m_prop] = (prev_tid, prev_prop)
  if (!commit) {
    pending.remove((t, m_prop))
  }
  try_deliver()
}
```

Listing 2.8: Server logic for RingWorld's deliver phase.

and server respectively.

Each ToR broadcasts the decision to the servers in its rack. Each ToR maintains a set of *suspected* servers, which are servers that have either crashed or are straggling. If a server has not responded with an `ACK` for the proposal by the time of `deliver`, the ToR suspects that the server has failed. The ToR then proceeds to take the suspected server *offline* by removing the server from its list of `children` (no further RingWorld packets are sent to the suspected server).

Upon receiving a `deliver` packet at the server, the server marks the proposal as deliver-

State	Type	Description
`predecessor_list`	Register array	Predecessor ToR IDs (by hop distance).
`predecessor_idx`	Integer	Current predecessor index (hop distance).
`recovery_counter`	Integer	Recovery ID, initially 0.
`decide_counter`	Register Array: Recovery id to ACK count	Recovery acknowledgement tracking.
`recovery_link`	Register array: Recovery id to (link_seq, link_t)	Previous sequence tracking.

Table 2.3: Configuration and state at each ToR for ToR failure recovery.

able and records its decision. The server then attempts to deliver any deliverable proposals. The server checks (i) if the proposal at the head of the `pending` log is deliverable and committed, and (ii) if the proposal's previous sequence number is also deliverable; if so the server moves the proposal from the head of the `pending` log to the `consistent` log. Any aborted messages are removed from the `pending` log.

2.3.3 Adding Servers to a Rack

In the deliver phase of RingWorld, the ToR takes any servers that have not responded in time offline. Doing so is necessary to prevent straggling servers from impacting RingWorld's latency or throughput. This begets the need to ensure servers can efficiently (re)join the rack. RingWorld has a *rack-local* protocol to add servers to the rack.

RingWorld's `server-join` protocol ensures that the joining server replicates all committed and `pending` proposals before it can join the rack. The joining server does so by communicating with other servers in the rack. The ToR also forwards all new proposals to any joining servers. The joining servers process all proposals normally and must acknowledge these proposals to prevent being taken offline again once they rejoin. However, they won't be able to commit any proposals until they have caught up on missing proposals owing to the previous sequence number undelivered check. Once the server is up-to-date, it can join the rack.

2.3.4 Handling Rack Failures

Racks are a failure domain in most datacenters, and a rack can fail for a variety of reasons including ToR failure or power failures to the rack. Since messages in RingWorld are forwarded along a logical ring of racks, a rack failure can pause processing and impact RingWorld's performance. We address this problem using RingWorld's recovery protocol. The recovery protocol rapidly repairs the ring by routing around the failed rack. It then ensures any proposals that were not completed (due to packets that were lost when the rack failed) are committed or aborted. Table 2.3 shows the additional state kept at the ToR for RingWorld's recovery protocol.

Ring repair

Detecting rack failures: A ToR in RingWorld sends periodic heartbeats to its logical successor ToR in the ring. The successor ToR uses a lack of `heartbeat` packets to detect a rack failure. Since heartbeats in RingWorld are sent and processed entirely by ToRs (that provide tight latency bounds on message processing) we can set very low timeouts for failure detection. In our experience, we found that we can set detection timeouts as low as 2ms without causing false positives.

Routing around rack failures: We start by describing how RingWorld handles a single rack failure, then briefly discuss the generalization to multiple rack failures.

Assume rack $t+1$ detects that rack t has failed. ToR $t+1$ sends a message to rack $t-1$ requesting it to change its successor from t to $t+1$. ToR $t-1$ changes its successor and starts forwarding all RingWorld packets to ToR $t+1$.

To reach ToR $t-1$, each ToR maintains a list of its predecessors and its current predecessor index in the logical ring which is updated upon detecting each rack failure. In the case of multiple rack failures, ToR $t+1$ would continue to contact $t-2$, $t-3$, and so on until it reaches the first non-failed rack.

Recovering incomplete proposals

When a rack fails some RingWorld packets might be lost. This results in proposals that were initiated but are not completed (committed or aborted), which in turn stalls the `pending` log processing at the servers. RingWorld must ensure that such proposals are either committed or aborted to make progress. Listings 2.9 and 2.10 show the pseudocode for the recovery process for the server and ToR respectively.

To do so, each server that received a proposal starts a `decided` timer (t_1) for that proposal. All timers are canceled when a corresponding `deliver` packet is received for that proposal. However, in the case of a lost packet (which could be `data`, `collect` or `deliver`) the `decided` timer will go off, initiating the recovery process. The server first attempts to recover the proposal locally in its rack by querying other servers in the same rack. The server starts a `recover` timer (t_2) at this point. Other servers in the rack only respond if they know a decision was reached for the proposal.

If the server receives no information regarding the proposal locally, the `recover` timer will go off initiating a global recovery. A `recovery` packet is sent serially around the ring where each ToR first checks whether any server has logged a decision. If any server has logged a decision, then the ToR switch sends an appropriate `deliver` packet to its successor rack.

```
callback timeout for timer t1, (t, m_prop, m) {
  % Initiate local recovery process
  send(parent, DECIDEDP(t, m_prop))
  start_timer(t2)
}

callback receive DECIDEDP(t, m_prop, server) {
  if can_deliver[(t,m_prop)] != null {
    send(server, DECIDEDV(t,m_prop, prev_tid, prev_prop, can_deliver[(t,m_prop])
  }
}

callback timeout for timer t2 (t, m_prop, m) {
  % Find the largest proposal smaller than (t, m_prop)
  prev_tid, prev_prop = find_latest(t, m_prop)
  % Initiate global recovery process
  send(parent, INIT_RECOVER(t, m_prop, prev_tid, prev_prop, m))
}

callback receive RECOVER_LINK(init_tor, rid, t, m_prop, recover_count, m) {
  if can_deliver[(t,m_prop)] != null {
    % If already decided tell the parent ToR the decision
    send(parent, LINK_RESPONSE(init_tor, rid, t, m_prop, prev_tid, prev_prop, true,
    can_deliver[(t,m_prop], recover_count, m)
  } else {
    % Replicate the proposal
    pending.insert((t,m_prop), m)
    can_deliver[t,m_prop] = null
    % Find the largest proposal smaller than (t, m_prop)
    prev_tid, prev_prop = find_latest(t, m_prop)
    send(parent, LINK_RESPONSE(init_tor, rid, t, m_prop, prev_tid, prev_prop, false,
    false, recover_count, m)
  }
}
```

Listing 2.9: Server logic for RingWorld's recovery process.

```
callback receive DECIDEDP(t, m_prop) {
  send(children, DECIDEDP(t, m_prop, server))
}

callback receive INIT_RECOVER(t, m_prop, prev_tid, prev_prop, m) {
  rid = recovery_counter
  recovery_counter += 1
  decide_counter[rid] = 0
  recovery_link[rid] = prev_tid, prev_prop
  send(children, RECOVER_LINK(id, rid, t, m_prop, 0, m))
}

callback receive LINK_RESPONSE(init_tor, rid, t, m_prop, prev_tid, prev_prop, decided,
     decision, recover_count, m) {
  if decided {
    % If already decided, circulate DELIVER around the ring
    broadcast(DELIVER(t, m_prop, prev_tid, prev_prop, can_deliver[(t,m_prop])
  } else { % If not, check how many servers have replicated the message, if enough
    servers have send it forward.
    if (recovery_link[rid] < (prev_tid, prev_prop)) {
      recovery_link[rid] = prev_tid, prev_prop
    }
    decide_counter[rid] += 1
    if (decide_counter[rid] == f + 1) {
      send(next, RECOVER(initiator, t, m_prop, recovery_link[rid][t], recovery_link[
    rid][m_prop], recover_count + 1, m)
    }
  }
}

callback receive RECOVER(init_tor, t, m_prop, prev_tid, prev_prop, recover_count, m) {
  if (initiator == id) { % Have completed a round around the ring, check if enough
    racks have replicated the proposal and circulate DELIVER
    if (recover_count > r) { % Commit
      broadcast(DELIVER(t, m_prop, prev_tid, prev_prop, true))
    } else { % Cannot commit, mark.
      broadcast(DELIVER(id, m_prop, prev_tid, prev_prop, false))
    }
  } else {
    rid = recovery_counter
    recovery_counter += 1
    decide_counter[rid] = 0
    recovery_link[rid] = prev_tid, prev_prop
    % Ask children for latest information
    send(children, RECOVER_LINK(initiator, rid, t, m_prop, recover_count, m))
  }
}
```

Listing 2.10: ToR logic for RingWorld's recovery process.

All RingWorld ToRs unconditionally forward `deliver` packets along the ring and broadcast them to servers in their rack. This is safe because `deliver` packets are idempotent. On the other hand, if a server has not logged a decision, it replicates the proposal and responds to the ToR. Once enough servers have replicated under a ToR, the ToR forwards the `recovery` message to its successor. Once the `recovery` packet makes its way back to the recovery initiating ToR, a `deliver` packet with the decision is circulated around the ring.

Split brain in RingWorld

RingWorld's ring could possibly split into multiple smaller rings. To maintain safety in such cases, only the ring with majority racks ($\geq r+1$) is allowed to commit proposals. At the time of `deliver`, RingWorld checks if $\geq r+1$ racks participated in the collect phase for this proposal. Due to this check, any smaller rings will not be able to commit any proposals guaranteeing safety.

RingWorld eventually repairs the ring such that a majority ring is formed. At the time of `deliver` a ToR detects it is no longer part of the majority ring when less than $r+1$ racks have participated in the collect phase. The ToR then circulates this information around the minority ring to update other ToRs. To repair the ring, when a ToR detects it is no longer part of the majority ring, the ToR takes itself offline and then initiates the ToR rejoin process (described below).

2.3.5 Connecting a rack to the ring (Rejoins)

RingWorld needs a rejoin process in case a rack fails or a split ring is formed. RingWorld's `rack-rejoin` protocol ensures that all ToRs point to correct successors and predecessors, and the new rack starts with the correct state as it rejoins the ring. We first describe the rejoin protocol and then describe a heuristic for finding the right place to join the ring.

Rack rejoin

The rejoining ToR contacts its chosen successor in the ring with a `ToR-join` packet. The successor updates its list of predecessors and circulates the packet around the ring, with each ToR doing the same. Once the packet arrives at the rejoining ToR's potential predecessor, the potential predecessor brings the rejoining ToR up-to-date. The predecessor provides its `proposal ID` and its list of `predecessors` which the rejoining ToR inherits. The rejoining ToR adopts the predecessor's proposal ID, as starting from any number lower would violate the proposal ID properties described in section 2.3.2. The rejoining ToR also shares the same set of predecessors as its predecessor. Once updated, the ToR becomes a part of the ring and can start normal processing.

Rejoin position

We present a heuristic for the rack to rejoin the ring at a correct place: the rack must join as the predecessor of the lowest ToR id of the majority ring. If no majority exists, use the lowest ToR id only. This will eventually result in a majority ring. To do so, the rejoining ToR contacts all ToRs and requests two things: i) its ToR id, and ii) whether the ToR believes it is part of the majority ring.

2.3.6 Ring size changes

RingWorld needs a view change to change ring size (changing ring size is different to rack rejoin). A view id must be maintained by each ToR, which is incremented every time the quorum size changes. The ToRs should only accept proposals with the latest view id. The view change requires two rounds around the ring to get a majority of racks to agree to change the quorum size.

2.3.7 Analysis

We now analyze the impact of various factors on the latency of RingWorld.

Number of racks in the ring: One would expect that RingWorld's latency would increase as we add more racks in the ring due to an increase in the ring size However, server processing is often slower and ToR-to-ToR communication latency is a small fraction of server processing times. Therefore, up to a certain threshold (where server processing times are larger than the entire ring's latency) increasing the number of racks in the ring does not impact the latency of the system. Beyond the threshold, increasing the ring size increases the latency of the system, although, not at the same rate as the number of servers (RingWorld has multiple servers in a single rack). The machine processing time can be improved by using various techniques such as kernel bypass or responding to the ACK from the NIC instead of going into userspace.

Number of servers in a rack: To ensure a proposal is sufficiently replicated, more than f servers in a rack must respond with an ACK to the proposal. All servers communicate with the rack's ToR in parallel. The latency of RingWorld, therefore, depends on the median response time of the servers to sufficiently replicate the proposal.

2.4 Evaluation

We evaluate RingWorld by addressing three questions: i) how does RingWorld perform with regards to throughput and latency in comparison to other ring-based and leader-based protocols? ii) how does RingWorld perform in the presence of server failures? and iii) how does RingWorld perform in the presence of ToR failures?

Implementation and experimental setup: We implemented RingWorld's ToR component on Tofino Barefoot Wedge 100BF-32X [31] in P4 and the client/server component in C++.

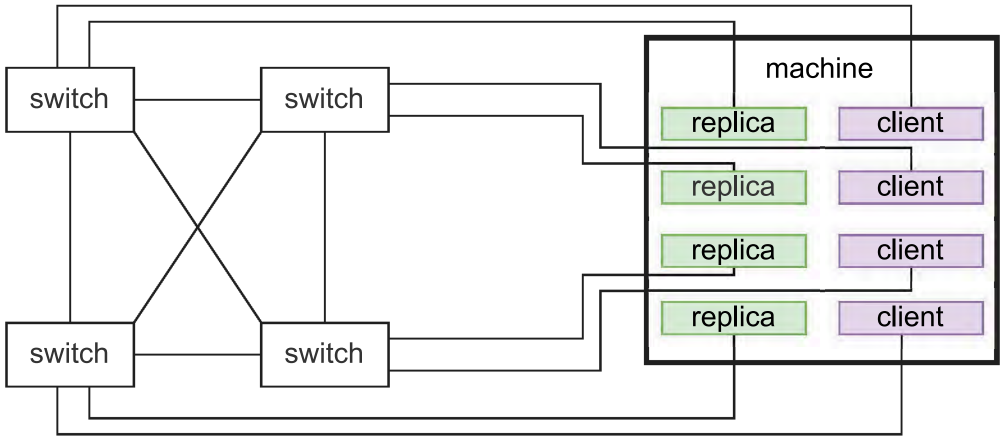


Figure 2.4: Testbed topology showing connections for a single server.

We set up a testbed of four ToR switches connected in a full mesh network. Additionally, we connected three 16-core Intel Xeon, 64 GB memory servers to our setup, with a single server's connections shown in figure 2.4. The other machines replicate these connections. Each server was equipped with eight 10 Gbps NICs, thereby each server running up to four replicas and four clients.

For RingWorld we connect twelve replicas with four ToR switches with each ToR having three replicas under it.

2.4.1 Throughput vs. Latency

To compare the performance of RingWorld we compare it to three other replication protocols: one ring-based replication protocol - similar to LCR [26] and two leader-based protocols - Viewstamped Replication [43, 51] (equivalent to Multi-Paxos) and NOPaxos [42]. Additionally, we evaluate it against an unreplicated system that does not provide any fault tolerance and provides a baseline for optimal throughput. We extended NOPaxos's original code and corresponded extensively with the authors to ensure a fair setup. The latency vs. throughput results are shown in figure 2.5 as we increase the number of clients and the clients are run in a closed-loop. In all cases, the throughput was bottlenecked at the replicas with 100% CPU utilization.

Unreplicated system comparison: RingWorld matches the throughput of an unreplicated system (∼85 Kops/s), as each replica communicates only with its own ToR, eliminating the throughput bottleneck introduced by extra message processing. However, the latency for RingWorld is higher than an unreplicated system as communication around the ring is needed to ensure a request is replicated across the system.

Ring-based replication comparison: For the ring-based replication protocol, we use 12 replicas connected in a ring. Figure 2.5 shows that while the ring-based system is able to

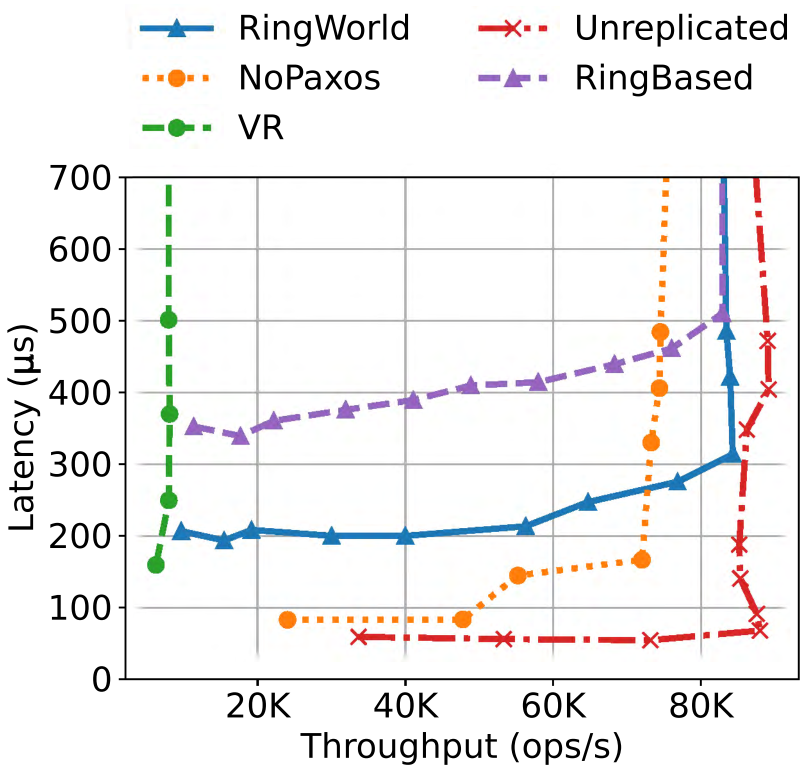


Figure 2.5: Latency vs. throughput comparison for RingWorld, VR, NOPaxos, ring-based consensus and unreplicated system.

achieve high throughput (due to limited communication between replicas), the latency is 2x higher for the simple ring-based protocol compared to RingWorld, even though both systems use the same number of replicas. This is due to the greater number of hops in the ring-based protocol's ring with slow server-to-server communication, compared with RingWorld's smaller ring size with faster ToR-to-ToR communication.

Leader-based replication comparison: For Viewstamped Replication (VR) and NOPaxos we use 3 replicas to replicate the requests. Figure 2.5 shows VR's throughput maxes out at a significantly lower ∼9 Kops/s, due to excessive messaging between the leader and replicas which makes the leader a throughput bottleneck. NOPaxos is able to achieve high throughput as ordering at the switch eliminates excessive message processing at the servers; the switch directly broadcasts a request to the replicas and replicas reply directly to the client. This also allows NOPaxos to achieve lower latency than RingWorld.

Exploiting a ring-based topology with a hierarchy of ToR and servers allows RingWorld to have a simpler failure recovery mechanism which we explore below.

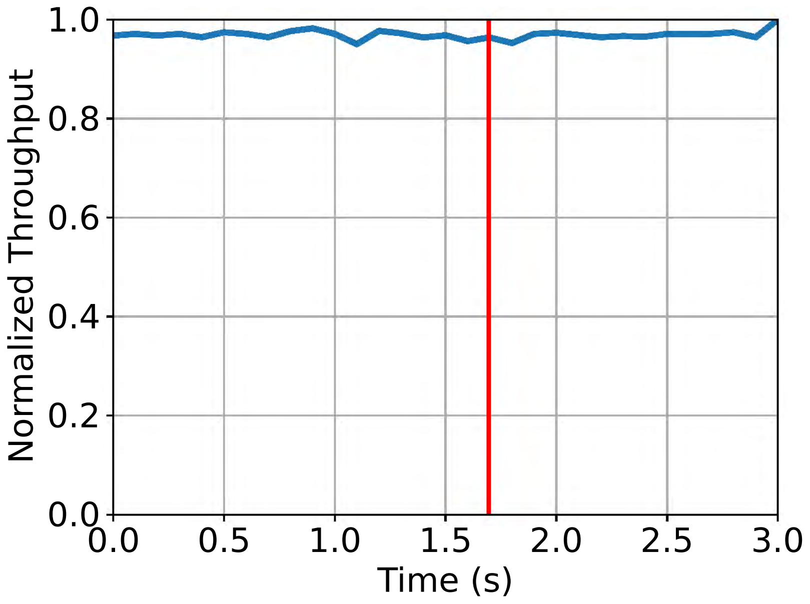


Figure 2.6: Normalized throughput for RingWorld during a server failure indicated by the vertical red line.

2.4.2 Server Failures

We now look at how RingWorld performs in the presence of server failures. Figure 2.6 shows there is zero downtime on the system's throughput as a server fails at the time indicated by the red line. A leader failure in VR and NOPaxos would lead to a view change protocol to re-elect a leader and would cause some downtime for the system. A server failure in a ring-based protocol would require either require a view change or routing around the failed server which also means some downtime would be experienced during server failures.

2.4.3 ToR Failures

We now look at how RingWorld performs in the presence of ToR failures. RingWorld needs to repair the ring to start processing requests again and catch up on any missed requests to ensure linearizability. The time to reach full throughput, therefore, depends on two things: (i) link repair and (ii) log resumption. We will look at their impact separately and then look at their overall impact on the system.

Link repair: We measure the link repair duration as we vary the granularity of heartbeat generation and failure detection. In our experiments, we set failure detection time to be four times the heartbeat generation time. To measure the link recovery time at ToR i, we record the time between the failed predecessor's $(t-1)$ last heartbeat and the new predecessor's $(t-2)$ first message/heartbeat. The results in figure 2.7 show that recovery

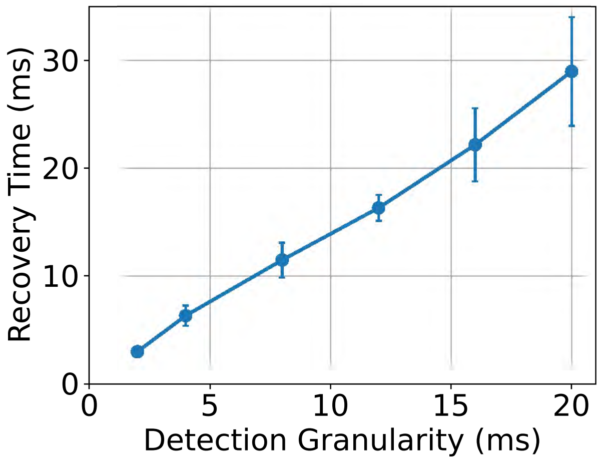


Figure 2.7: RingWorld's link repair time after a ToR failure as a function of detection granularity.

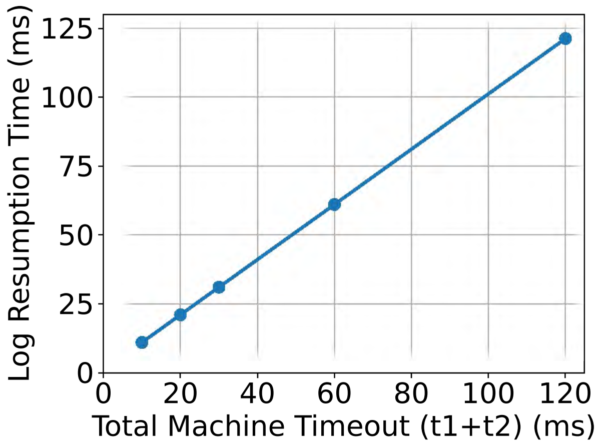


Figure 2.8: RingWorld's log resumption time after a ToR failure as a function of sum of `decided` and `recover` timeouts ($t_1 + t_2$).

time is within a few milliseconds of the detection granularity and grows linearly with the detection granularity.

In terms of bandwidth overhead, our heartbeat packet contains the Ethernet and RingWorld headers, which are 14 bytes and 7 bytes respectively. However, the Tofino packet generator imposes a minimum size of 60 bytes which we fulfill by padding the remaining header with zeros. We do not need to generate heartbeat packets when we have actual request packets to send.

Log resumption: The messages at the servers must be delivered in order to maintain linearizability and message delivery is paused if a message that ought to be delivered first is missing. The resumption of message processing depends on the two server timeouts: `decided` (t_1) and `recover` (t_2). In our experiments, we vary the two timeouts, while keeping failure detection granularity at 2 ms. We measure resumption time as the time between a `data` message was sent out before ToR failure and the time when its corresponding `deliver` message is received after ring recovery. Figure 2.8 shows the log resumption time as we vary the two timeouts. The plot shows the sum of the two timeouts on the x-axis with both t_1 and t_2 timeout values being equal. The plot shows that the resumption closely follows the sum of the two timeouts and increases linearly as the sum of the two timeouts increases.

ToR recovery: Taking a holistic look, figure 2.9 shows the normalized throughput of the system after we fail a ToR at the time indicated by the red line. In this experiment, the detection granularity is set to $2ms$, and the `decided` (t_1) and `recover` (t_2) timeouts are set to $5ms$ each. The results show that the throughput drops partially for 10-15 ms before bouncing back. In contrast, NOPaxos relies on an external SDN controller to recover from switch failures and the downtime will depend on the controller [42].

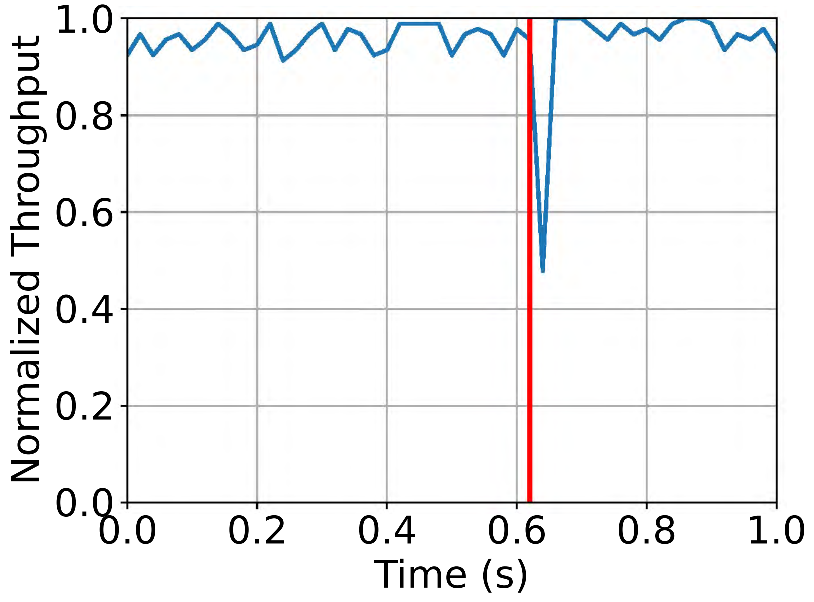


Figure 2.9: RingWorld's normalized throughput during a ToR failure with 2ms failure detection timeout and 5ms `decided` and 5ms `recover` timeouts.

2.5 Related Work

Many classical log-based consensus algorithms use a single leader to order requests *e.g.,* MultiPaxos [41, 39], Viewstamped Replication [51, 43], Raft [52], and ZAB [34]. However, they suffer from poor performance (latency and throughput) due to the leader being a bottleneck. As a result, many optimizations to improve throughput and latency have been proposed. In this vain, Fast Paxos [37], SpecPaxos [54] use approximate network ordering, and NOPaxos [42] uses programmable switches to guarantee network ordering to improve latency and throughput. Hovercraft [35] uses in-network programmability and transport designed for datacenters. Mu [1] utilizes RDMA to provide microsecond latencies, while Derecho [32] uses it to provide high throughput. P4xos [15] implements Paxos in programmable switches to improve throughput and latency. However, all of these protocols require complicated failover procedures to recover from leader failure or are dependent upon a separate replicated entity *e.g.,* SDN to recover from failures.

In contrast, Mencius[44], EPaxos [47], and Rabia [53] take a leaderless approach to achieve consensus. A leaderless approach eliminates the single leader as the bottleneck and allows these protocols to achieve higher throughput. However, they exhibit increased latency in certain cases. Mencius partitions the log amongst replicas, but its latency suffers in the case of straggling replicas. EPaxos requires indicating dependencies and can be slow in the case of dependency interference. Rabia uses randomized consensus and requires more communication to achieve consensus.

Most closely related to RingWorld is the work on Ring- or chain-based consensus algorithms which organize the replicas into a ring (or chain) and replicate requests sequentially, *e.g.,* LCR [26], RingPaxos [45], Chain Replication [68], CRAQ [65] and NetChain [33]. Each replica only communicates with its neighbors, providing optimal throughput. However, the latency of these systems scales linearly with the size of the ring. They also require either complex fail-over protocols or depend on a separate replicated entity to recover from failures. Another ring-based algorithm, Carousel [25] provides a simple failure recovery protocol in case of failure, but still suffers from scaling latency as the size of the ring grows.

RingWorld utilizes programmable switches to arrange racks in a ring. Exploiting the ring topology allows RingWorld to achieve optimal throughput and a simpler failure recovery procedure. The hierarchy of ToRs and servers in a rack (with server communication under a ToR happening in parallel) prevents RingWorld's latency from growing with the number of servers in the ring.

2.6 Conclusion

In this chapter, we present RingWorld, a protocol that leverages programmable switches to adapt a ring-based approach to achieve consensus. RingWorld arranges racks in a ring; programmable ToR switches are connected to form a logical ring with multiple servers under each ToR. Processing is partitioned between fast-path ToRs and slow-path servers. ToRs maintain consistent ordering between requests, and forward requests around the ring, whereas servers maintain a consistent log. The response latency for RingWorld does not grow with the size of the ring (until the ring becomes really large). This is due to the ring propagation time being lower compared to server processing times.

RingWorld is able to achieve optimal throughput, comparable to an unreplicated system, and latency comparable to existing RSMs. Additionally, exploiting a ring-based topology allows RingWorld to have a simple failure recovery process with low downtime. Server failures do not impact request processing and ToR failures do not need an expensive view change protocol, but only communication between local neighbors to repair the ring.

Chapter 3

Fault Tolerance at the Edge

We are now moving computation to the edge
But the failure case has me on the ledge
But in this chapter, we do solemnly pledge
that about correctness you must no longer hedge

Edge computing has recently entered the hype cycle, but it is important to remember that, with Akamai being founded in 1998, we have had edge computing in the form of CDNs since soon after the Internet went public. In recent years, however, we have seen the emergence of a new and more varied generation of edge computing, with the likes of Apple, Google, Netflix, and other major content providers establishing their own edge infrastructures, and commercial offerings, such as AWS Lambda@Edge, Cloudflare Workers, Akamai Cloudlet, Fastly Edge Compute Platform, and Azure Edge Functions, allowing tenants to deploy computation at the edge.

This nontrivial computation is being placed at the network edge[1] for many reasons, including: lower-latency responses to clients (such as in games and content provision), lower bandwidth demands on the backbone (such as in IoT and some video applications), and increased privacy where the edge handles information that clients do not want seen by the backend server (which arises in some video and IoT applications).

Earlier uses of edge computing such as CDNs were either stateless or soft-state, so their correctness and reasonable performance did not depend on the edge retaining any state.[2] However, many of the new uses of edge computing – such as games, video analytics, and IoT – are *strongly stateful* in the sense that either the correctness of the application, or

[1]In this work, we use the term *edge* to describe any application-level processing node that is placed between a client and a server. Such an edge could be placed, for example, in a branch office, an ISP central office, or a factory floor.

[2]Of course, the *raison d'être* of CDNs is to cache state (*i.e.,* content), but if that state were deleted it would merely result in a cache miss and the request would be forwarded to the origin site. While the resulting performance is not optimal, it is still within normal operational bounds since cache misses are not rare events.

its ability to achieve reasonable performance, requires that the edge state be maintained. Applications are strongly stateful if when the edge state is lost: (i) the application correctness requires that the state be reconstructed and (ii) that reconstruction (if possible at all within the application's normal operation) incurs a substantial performance penalty. Being strongly stateful poses a problem when an edge fails and another edge is available (so that connectivity can be re-established), but the state from the failed edge is not present on the new edge. We address this challenge in the context of client-server network services.

Typically the client-server paradigm is built around the notion of a client *session* (*i.e.,* a client's ongoing interaction with the server). Inserting a strongly stateful edge to improve the performance of a client's session turns the client-server paradigm into a client-edge-server paradigm. With the original client-server paradigm, *fate sharing* is assumed to exist between the session and the client and server, such that if either the client or server dies, the session is terminated. While there are multiple techniques that have been developed to improve server resilience (*e.g.,* using replicated state machines) and client resilience (*e.g.,* using multihoming to allow clients to survive some types of network outages), the lifetime of a session is still fundamentally tied to both the client and server being available.

In the new client-edge-server paradigm with a strongly stateful edge, the session's fate is now shared between all three entities, in that if any of the three stops functioning, the session cannot continue (at least not without a significant performance penalty). While the session's reliance on the client and server is inherent, the reliance on the edge is problematic since an edge failure can terminate a session even when the client and server remain alive and another edge is available to provide connectivity. We note here that the problem of fault-tolerance is also relevant to the case of client mobility; as clients move, they may need to change the edge to which they connect. While there are techniques for smoothly moving the edge state to follow the client, in the worst case (where such state migrations are not implemented or fail to complete), this poses the same challenge as an edge failure.

To make our discussion of how best to provide fault tolerance for strongly stateful edge computing more concrete, we present a video analytics application as a motivating example; this example did not come from our lab, but instead was brought to us by the team who has deployed this application in production and had struggled with the problem of edge failures. Many video analytics frameworks rely on edge computing to minimize the amount of data transferred from a camera to the backend servers (typically in datacenters). The savings can be significant because many video frames contain little or no actionable data, so these frames can be safely filtered. However, this filtering often depends on knowing what information has previously been sent to the backend servers. If the state about this previously sent information is lost, then the video analytics application must stop filtering at the edge (or stop forwarding traffic completely) until it can restore enough shared state between the edge and the server so that filtering can resume. As we explain later, for the application we consider, this can interrupt video service for several minutes.

To provide uninterrupted service for this and other strongly stateful applications, we need a mechanism to recover from edge failures that ensures correctness and provides reasonably good performance. We propose a general purpose solution that uses message replay and checkpointing. These are, of course, not novel techniques, but our main contribution is to adapt these techniques to the edge context and thereby provide an effective solution for the real problem of edge fault-tolerance.

To achieve this solution, we first identify a consistency guarantee that we refer to as *output message consistency* that applies to the client-edge-server paradigm. We then describe the design and implementation of CESSNA (Client-Edge-Server for Stateful Network Applications), which is an application framework that provides output message consistency for each application session. We designed CESSNA to require minimal modifications to application logic, and thus any client-edge-server application can readily adopt CESSNA, and be tolerant to failures at the edge.

We have implemented two prototypes of CESSNA, using two different sets of technologies: The first, Container Isolated CESSNA (CI-CESSNA) uses Docker, an off-the-shelf Container platform, and provides an Edge API based on Python. This version is designed to minimize the number of changes required when adopting CESSNA, but this ease of adoption comes at a slight increase in overheads. The second, Software Isolated CESSNA (SI-CESSNA), requires applications to make use of specialized CESSNA data structures, which in turn allow us to reduce application overhead. We have implemented versions of SI-CESSNA for both Rust and C#. In addition to measuring the SI-CESSNA performance with our video analytics example, we also deployed both the SI-CESSNA and CI-CESSNA implementations in multiple locations worldwide and ran several other example edge applications. We discuss our correctness guarantees, and present experimental results to show that CESSNA provides these guarantees with minimal performance overhead in the absence of failures, and reasonable recovery times when there are failures.

3.1 Background and Related Work

Before delving into our design, we first describe some relevant background about how the edge is currently being used and about the various mechanisms now used to provide fault tolerance.

3.1.1 Current Edge Computing Efforts

We start by briefly discussing the various forms of shared edges (*i.e.,* edge computing services offered to tenants) currently available, along with a quick review of special-purpose edge computing.

Content delivery networks (CDNs): CDNs such as Akamai, Cloudflare, Fastly, Azure CDN, and Amazon CloudFront represent the earliest attempts to use resources at the network edge in order to improve application performance. Caching content in the network benefits three parties: the client's ISP, who now has to transfer a smaller quantity of data over the network core; the content provider, who can more easily scale to larger number of clients;

and clients, who experience lower latency when accessing content. As a result, CDNs have been widely adopted, and form a core component of the Internet infrastructure. CDN caches however offer little in terms of general computational capabilities.

Cloudlets: Cloudlets [58, 57] is an academic project in which computation is performed on servers at the edge of the network. Cloudlets were originally envisioned to augment the capabilities of mobile clients, but the Cloudlet computational model is general and has been adopted by a few companies including Akamai [3]. The Cloudlet design places four main requirements on these offerings, one of which explicitly requires that the edge only contain soft-state – state whose loss does not impair the correctness of the application. The authors state that this requirement simplifies management, in particular simplifying the task of handling client mobility and failures. While some recent efforts [27, 7] have looked at using VM live migration in order to reduce the impact of lost state during client migration, these efforts assume that neither the old edge nor the new edge has failed, and provide migration times on the order of minutes.

Serverless edge offerings: These offerings – such as AWS's Lambda@Edge, Azure Function on IoT Edge, and Cloudflare Workers – allow developers to write serverless applications that are executed at the edge. Similar to current serverless offerings, most of these services require the use of cloud storage services such as blob stores and databases to store state. Azure's Durable Functions [19] are an exception to this rule, allowing functions to persist state locally. In order to do so, Durable Functions require developers to use short-lived functions called *activities* in order to manipulate state. An orchestrator, which can be customized by the developer, logs the sequence of activities that have been executed. Durable Functions replay activities in order to recover from failures or reconstruct state that was lost for other reasons. Durable Functions therefore do provide mechanisms for recovering from edge failures; however, the failure recovery process requires applications to be restructured in order to log modification to individual state elements. CESSNA, by contrast, adopts a more traditional approach to checkpoint and replay, treating the entire edge process as a single entity, thus minimizing the application changes needed to provide fault-tolerance.

Single-application edges: There is a growing trend for inserting computation into application-specific edges (in some cases replacing functionality previously located in the cloud). Examples include automation technology on factory floors, and smart-camera and other video analytics applications [48, 70, 55, 29].

3.1.2 Fault Tolerance and Message Replay

There is a long history of distributed systems that rely on replication, checkpointing, and message replay to provide fault tolerance. In situating our design within this literature we consider four types of work:

State Machine Replication: Replicated state machines [60] and closely related work such as viewstamped replication [51] and virtual synchrony [6] have long been the standard approach to providing fault tolerance for many services. The core idea used by these techniques is to deploy several replicas of a service, and then execute messages in the same order at each replica. The main challenge when using these techniques lies in ensuring that messages are processed in the same order at all replicas, and this is addressed either through the use of consensus protocols such as Paxos and Raft, or the use of group communication primitives like atomic broadcast. Standard results in distributed systems [11] show that both of these approaches are equivalent, and most modern implementation build on consensus based approaches.
Primary Backup Replication for VMs and Processes: Auragen 4000 [8], Remus [14], and other systems have relied on VM and process replication [59] in order to provide fault tolerance. These systems run multiple replicas of the same service, and treat one of these replicas as the primary. All external inputs including messages and interrupts received by the primary are assigned a processing order and sent to the replicas. This ensures that all replicas agree on the external order of events, and replicas can take over when the primary fails. This approach poses two main challenges: first it requires multiple active replicas; second, in order to meet consistency requirements when handling failures, these systems require that the primary replicate inputs before releasing any outputs. The former increases resource requirements, while the later impacts system latency and throughput.

Record and Replay: Record and replay systems such as ReVIRT [18], SMP-ReVIRT [17], and FTMB [62] are designed to reduce the resource requirements of the previous techniques by eliminating the need for active replicas. These systems execute a single primary as a VM, and periodically checkpoint the primary's state. These systems also record all external inputs between checkpoints. When the primary fails, these systems recover by first restoring a VM from the last good checkpoint, and then replaying all external inputs in order to produce a replica with the same state as the primary. While the use of checkpoint and replay eliminates the need for active replicas, it in turn requires the use of an additional agent, which must be located in a different failure domain, that records all inputs to the primary. Existing work assume that this agent is run on a different server than the primary in order to meet this requirement.

Why do these techniques not suffice for the edge? We assume that each edge location has a limited number of servers; this is both due to economic necessity (there are many more edges than cloud datacenters) and limitations of available space, power, and cooling. Thus, techniques such as Remus would not be appropriate for edges since their use would require doubling the required compute resources. Moreover, we assume that one common failure model for the edge is a complete site failure (as these edge sites are often small and not equipped with multiple power sources and the like). Given this assumption, recovering from an edge failure often requires failing over to a replica at a different edge, and these edges might only be connected via wide-area networks. Prior work [27] has made similar assumptions when handling client mobility. As a result, neither state machine replication nor primary-backup replication are suited to the edge use case: recent works including WPaxos [2] and Mencius [44] report that consensus protocols when run on the wide area impose per-message latencies of 100-200ms and can only support 10,000 operations per second or less. While, some of the recent literature on wide-area replication [20, 63, 46] have built on conflict-free replicated data types [61] (CRDTs) in order to address these performance limitations, adopting CRDT based techniques requires changes to application logic and a restructuring of application state, and hence these techniques cannot be used with general applications.

Given these various limitations, for CESSNA we have chosen to adapt the record and replay based mechanisms for the edge. This requires designing CESSNA so we can survive the failure of an entire edge location, which obviates most traditional record and replay designs which store the recordings nearby. Instead, we rely on the client and server for message logging since this maximizes the extent of fate sharing; recovery is possible if and only if the two endpoints are up and connected, which is exactly the fate-sharing semantics that client-server applications have (and which traditional record and replay solutions do not achieve).

We later define the sufficient correctness requirement for CESSNA (in Section 3.2.2), but here we just note that this consistency model is different than previously discussed models in the sense that it is defined per session (and not per application, or for a specific request), and while it is stricter than eventual consistency, it allows for recovery using replay based mechanisms, despite the ordering problem that the edge setting presents. We further elaborate on this aspect in Section 3.3.

Relation to other consistency models: Previous work has looked at several consistency models for distributed data stores. This includes a variety of weaker models such as AMBROSIA [24], Bayou [66], and the proposal by Nightingale et al [50]; and reformulations of existing models [13]. Consistency models in distributed data stores dictate when *updates are visible* to different clients. While we similarly describe a consistency model in this chapter, the goal of our model is to reason about when *updates are stable*, *i.e.,* about the state of an application after failures. These different goals render these models incomparable in both efficacy and performance.

3.2 Our Approach

3.2.1 Computational Model

Our design imposes no restrictions on how clients or servers are built. In particular, it does not preclude the use of mechanisms at the application level for recovering from client or server failures, or the use of replication or other techniques to increase the resiliency of the server or client. Our focus in this chapter is preserving correctness after a change at the edge due to mobility or failure. Thus, in what follows we assume a single (logical) client and a single (logical) server.

We assume that in the applications we consider both the client and the server can send packets to the edge, and that the edge can send packets to both the client and the server. A client starts a *session* when it first contacts an edge. All messages between this client and the edge, and between the edge and the server that corresponds to this client session, are considered part of this session. In our model, we assume that a session can either be terminated explicitly (being torn down by client or server) or implicitly (*i.e.,* due to client failure, server failure, or when no functional edge is reachable). We also assume that communication between clients, edges, and servers is over TCP connections, so that message delivery is in-order and all lost messages will be retransmitted.

The Edge We assume that the edge is stateful on a per-session basis: that is, a new edge process (or set of processes) is instantiated to handle each client session. We assume that the edge state for each session depends on the data sent and received within the session, on the order in which messages are processed at the edge, and on non-deterministic events such as timers and thread scheduling. Further, we require that the edge application software (*i.e.,* the code run at the edge) be designed so that state updates are atomic and each message is processed using only one version of the state.

We focus on providing edge fault tolerance on a per-session basis. We make no assumption on the number of edge nodes that can fail. For example, a single edge node can fail, but entire edge site failures are also possible. We would like to provide survivability such that as long as at least one edge node is available, not necessarily geographically close to the failed edge, the session can be recovered. When a single edge node fails, we would like support recovery mechanisms that recover quickly to a physically co-located edge node. However, in the case of a complete site failure, in order to ensure survivability, we would like support recovery to a completely different site.

Servers We place no restrictions on the behavior of the server. Similar to the existing client-server paradigm, the server can service multiple clients simultaneously.

Clients Similarly, we place no restrictions on the behavior of clients. We assume that clients can be mobile, and as a result they might connect to different edges over time. Note, however, that the client-server paradigm assumes that clients do not interact with other clients directly, but instead all such interactions are mediated through the server. Thus, to preserve this, we do not consider interactions between clients at the edge, and assume that all state at the edge belongs to a single client-server connection.

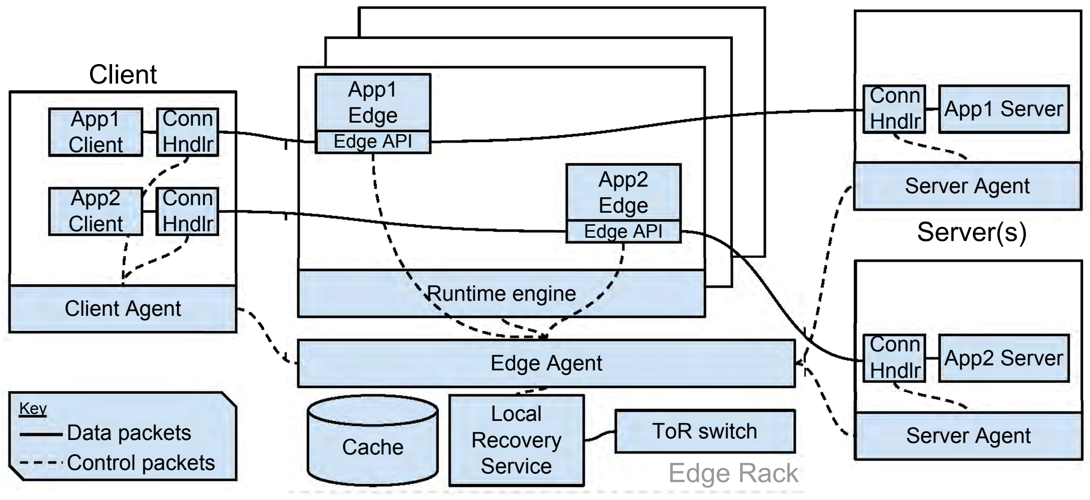


Figure 3.1: The general design of our framework.

3.2.2 Consistency Requirement

We focus on the case where a client is initially connected to one edge, but then must switch to another due to the failure of the first edge or because of client mobility. Our goal is to ensure that the processing of messages (from either client or server) at this new edge is consistent with what would have happened at the old edge if it had continued functioning.

We formally define the required correctness guarantee as *output message consistency*: messages emitted by a correctly recovered edge must be consistent with messages sent by the original edge before the failure and received by the client or server. This means that the recovered edge must be restored to the last *committed state* – the state at the last time the failed edge emitted a message that was received by either the client or the server. Note that our consistency guarantees do not require that edges be restored to the state right before failure (or mobility), only to the last committed state. This is sufficient for achieving output message consistency because only the last committed state is visible to the client or server.

Moreover, we want to achieve this level of consistency while maintaining reasonable performance and ensuring both *transparency* (client and server logic should not need to be changed to support edge recovery, though the edge logic might need to be aware of the recovery mechanism) and *survivability* (edge failure does not kill the session, as long as there exists a reachable edge to fail over to; this edge can be physically co-located with the failed edge or in a completely different site).

3.3 CESSNA's Design

In this section we describe the design of CESSNA (illustrated in Figure 3.1). The line of reasoning behind the design is straightforward. When moving to a new edge, we need to ensure that, before this new edge processes new packets from the client or server, it has established the last committed state of the previous edge (which is the old edge's state when it sent the last message that reached the client or server). The naïve approach to achieving this involves replaying at the new edge all the messages processed at the old edge before it reached its last committed state. However, the information for doing so is dispersed: the client knows which messages it has sent, the server knows which messages it has sent, but only the edge knows which of those messages were received, and in what order they were processed. Moreover, while the edge knows which messages it has sent, only the client and server know which of those were received. In addition, only the edge can capture the state needed to resolve nondeterminism in its processing of messages. All of this information must be effectively and efficiently combined to accomplish the faithful restoration of the last committed state in the new edge.

We also must deal with some more practical issues. For instance, for a mere ordering of the messages to be sufficient, the processing of messages must be atomic (in short, the processing of messages must be serializable). Also, the naïve approach would result in an infinitely growing set of messages that need to be replayed. We use checkpoints in order to truncate the sequence of messages that need to be replayed during failure recovery.

These are the issues that are addressed by CESSNA. The resulting design has two main pieces – the edge platform, and the client/server platform – which we now describe. Before doing so, we note that our data plane protocol (to be described later) provides reliable, ordered, message-oriented delivery.

3.3.1 Edge platform

General Properties: The edge is the only entity that knows in what order it received messages and what sources of nondeterminism affected the processing of those messages. We address the first with an *interleave log*, which records the order in which messages were processed by the edge application. However, many common programming patterns can introduce nondeterminism (or at least what appears to be nondeterminism from the perspective of the inputs). The most obvious of these is any program which utilizes the results from a random number generator, but can also include interactions with the OS scheduler (*e.g.,* due to threading), and using timekeeping functions. The CESSNA edge platform captures these external inputs as *events* that are added to the interleave log, so that when replayed at the new edge it can arrive at the same state as the old edge. To properly capture all such sources of nondeterminism, we require that edge applications utilize our edge API, which we show in Table 3.1 and describe below.

Every time a message is sent out from the edge, it contains the incremental update to the interleave log, so with every message from the edge, at least one of the client and the server has been sent the most up-to-date interleave log.[3] As we argue more formally in

[3]Because we use strong message ordering semantics on the client-edge and server-edge connections, where messages are read in order, we can use incremental logs to reconstruct the full logs.

Method	Description
`send_msg_to_client(msg)`	Send a message to client
`send_msg_to_server(msg)`	Send a message to server
`cache_read(obj_name, [func])`	Read an object from cache
`set_timeout(func, time)`	Start a timer
`random()`	Generate random number
`now()`	Retrieve current time
`lock_acquire(lock)`	Acquire a lock
`lock_release(lock)`	Release a lock

Table 3.1: Methods provided by the Edge Application API.

Section 3.4, this is the key to guaranteeing output message consistency. The CESSNA edge framework adds the CESSNA data plane header to all outgoing packets, and each packet carries incremental logs.

Since the interleave log can grow arbitrarily large over time, which would require the playback of an arbitrarily long set of messages, we will periodically take a checkpoint of the current state so that previous portions of the interleave log can be truncated. The taking of a checkpoint is entered into the interleave log as an event, so one can know which messages were processed after each checkpoint. The checkpoint is then sent to the client or server so they can use it in the recovery process.

Checkpoints can be taken with tools included in Docker [16], KVM [36], and VMware [72]. An alternative approach is to build edge application code using *explicitly checkpointable data structures*. We prototype both approaches.

In order to identify incoming and outgoing messages, and to treat messages atomically, we designed the CESSNA data plane protocol, which is a simple layer-7 encapsulation protocol. The protocol's header contains a sequence number and allows us to attach any updates to the interleave log. We wrap all messages with this header, which precedes any layer-7 payload.

We equip the edge with an *edge agent*, which is the control plane orchestrator of a single edge. It communicates with agents in the clients and in the servers, and provisions an edge application instance for each new session. The term *edge* here is flexible, and may refer to a single physical machine providing edge services, a single rack of such machines, or several racks – all of which can be managed by a single edge agent.

Edge API: We designed an edge application API that provides the methods shown in Table 3.1. In addition to the methods described in the table, the API also provides the following three event handler methods, which are to be overridden by the application. `accept_connection` is called when a client connects. The receipt of messages from the client or server result in calls to `recv_client_msg` or `recv_server_msg` as appropriate.

The underlying framework maintains connections to both the client and the server. It runs a control loop that continually reads available data from these connections and triggers the appropriate event handler as described above. While doing that, it maintains the aforementioned interleave log.

Based on the application configuration, the underlying framework may, after it finishes handling an incoming message, request that the edge agent take a checkpoint of the application. The framework waits until the checkpoint is taken before reading and handling the next message.[4]

Nondeterministic operations: We introduce the `now` and `random` methods for retrieving the current time and random number generation respectively. When not being used for replay, these methods call the corresponding underlying runtime's function, store the result in the interleave log with identification of the calling thread, and return the result. We allow timers via a `set_timeout` method. When the timer expires, the user-provided function is invoked by the main thread immediately after the current message (if any) is finished being processed. The sequence relative to other events/messages is stored in the interleave log.

Thread scheduling is another source of nondeterminism. To capture this, we require that threads be created using our API, which wraps the underlying runtime's threading capabilities but manages thread identifiers and synchronizes thread startup. The event handlers for accepting connections and reading messages are only invoked from the main thread of a CESSNA edge application (though it can then dispatch messages to other threads). Any thread is free to invoke other API methods, including the ones used to send messages. If threads share data, they must use explicit locks (mutexes). Our `lock_acquire` method logs every acquire operation to the interleave log after successful acquisition. Upon replay, the locks maintain the same order of acquisition. We note that our API could be extended to include other types of concurrent data structures and synchronization tools using the same technique we use for locks.

Edge Cache: Each edge runtime has a shared content cache that can be used by multiple instances of the same edge application. In order to guarantee the correctness of a replay process, the cache is read-only for edge applications. As is typical behavior for edge caches (*e.g.,* in CDNs and Cloudlets [57]), the cache fetches missing items from the server, so every read operation to the cache returns a result.

3.3.2 Client/Server Platform

There is a very little difference between a client and a server in our design. A host, either client or server, is just an application running atop our host platform, which manages communication with the edge. Our host platform consists of a host agent and a connection handler. The host agent is responsible for edge discovery, establishing a session with the edge and its management. In case of an edge failure or client migration, it is also responsible for

[4]We return to the issue of checkpoints in Section 3.6, where we note that checkpoints can be incrementally computed, greatly reducing the time the framework needs to wait.

reestablishing the session through another edge. The connection handler encapsulates outgoing packets to add the CESSNA data plane header, buffers these messages, decapsulates incoming messages, and stores the received interleave log.

In client-edge-server applications, the client does not connect directly to a known server, but to an edge – potentially one of many. Determination of which edge to connect to may depend on the application, client and edge locations, and other factors. Existing service discovery techniques, such as DNS or IP anycast [23, 69] can be used for this purpose.

3.3.3 Recovery

CESSNA supports multiple forms of recovery, but we begin with the most basic, where recovery is at a new edge that is remote and cold (leaving discussion of local and hot recovery variations until later in the section). These different recovery mechanisms are complementary, applying to different failure scenarios (*e.g.,* server or site failures), and can be deployed in parallel, but all ensure output message consistency.

First, assume that the client notices that the edge has not been responsive, or is otherwise malfunctioning. The client then initiates a recovery process by sending a message to a new edge, which it knows about via some discovery process (as mentioned above); from this point onward, the client ignores all subsequent messages from the old edge. This initiation message contains the sequence number of the client's most recent edge checkpoint (if it has one), its current version of the interleave log (suitably truncated due to the checkpoint), and the set of messages sent by the client that are contained in that log (and any more recent messages).

The new edge then sends a message to the server notifying it of the recovery process, and the server responds with the sequence number of its most recent checkpoint, its current version of the interleave log, and the set of messages sent by the server that are contained in that log (and any more recent messages). Once it is notified of this recovery process, the server ignores all subsequent messages from the old edge. The new edge then retrieves the most recent checkpoint from the client or server, verifies its integrity, and selects the most recent version of the two interleave logs. The new edge now restores the checkpoint and replays the messages and nondeterministic operations from the most recent interleave log in the proper order. At this point, the new edge has achieved the last committed state. It then replays any additional messages that it was sent by the client and server (interleaving them arbitrarily) and announces to both the client and the server that it is ready to handle new messages.

The message replay process is described in Algorithm 1. There are two subtle points. The first is that replaying the same inputs at the new edge will prompt the new edge to produce the same outputs as the old edge did when responding to these events originally. As the CESSNA dataplane protocol uniquely identifies all messages by sequence number, the recipient can trivially discard such duplicates. However, for efficiency, the algorithm simply filters them during replay. The second subtle aspect is the treatment of nondeterminisitic operations. During replay, a calls `random` or `now` do not generate new values, and we instead return values from the interleave log. Observe that if a value is not logged in the interleave

Algorithm 1 Edge application replay algorithm (main thread)

```
    Input:
    - client_order - interleave log known to client
    - server_order - interleave log known to server
    - client_msgs - client's message replay
    - server_msgs - server's message replay
    - checkpoint_seq - sequence number of the restored checkpoint
    - mrc - messages received by client
    - mrs - messages received by server
    - threads_wait_evt - an event on which all threads but the main
      thread are waiting if trying to invoke an API method. Initially
      this event is set (so threads wait).

 1: Initialize client and server connections
 2: Trim client_order, server_order to start from after the entry of checkpoint_seq, if exists, or set to []
    otherwise.
 3: ordering ← longest(client_order, server_order)
    % Take the longest interleave log provided by both the
    % client and the server, starting from after the checkpoint.
 4: ordering[thread_id] ← split(ordering)
    % Per-thread interleave log
 5: out_ordering ← merge(mrc, mrs)
    % Merge log of outgoing messages. Use this log to reorder
    % outgoing messages.
 6: threads_wait_evt.clear() % Let threads invoke API calls
 7: for each idx in ordering[main_thread] do
 8:     if idx is a timer event then
 9:         Mark timer as already executed
10:         Process timer event immediately
11:     else
12:         Let msg be the message with index idx in either client_msgs or server_msgs
13:         Replay msg: if replay emits messages, suppress those seen by client or server (based on mrc, mrs).
            Reorder emitted messages based on out_ordering.
14:         If replay calls random or now, find result in
            ordering[main_thread] and return it. If not found, generate new result.
15:     end if
16: end for
17: Replay all remaining messages in client_msgs and server_msgs, in any interleave order, without output
    suppression. Also handle waiting events.
18: Wait for all threads to finish going over their ordering
19: Start processing new data from client and server
```

log, then the operation was not successfully completed before the last committed state, and we can normally reexecute this operation. Similarly, callbacks for timers and cache reads are replayed in the order logged without delay.

CESSNA also maintains the same order of lock acquisition as indicated in the interleave log. When an edge application is multithreaded, the interleave log stores the ID of the thread corresponding to each entry for all types of entries. Upon replay, the recovery algorithm bases the order of outgoing messages on this information, and blocks threads when necessary to produce the same ordering of output messages as originally.

Other recovery scenarios: The above describes how we recover in the most general *remote recovery* case, when the location of the new edge was arbitrary and all state was stored at the client and server. However, for better performance, we simultaneously support *local recovery*, which is when the new edge is close to the old edge (perhaps even in the same rack). Local recovery uses a *local recovery service*, which is responsible for storing checkpoints, message logs, and interleave logs for multiple sessions. This service can be deployed per physical machine, or per rack of multiple machines. The local recovery service has a direct connection to the top of rack switch's tap port, so it can reconstruct the corresponding TCP sessions and extract incoming and outgoing CESSNA data plane messages to construct its local copy of message logs and the interleave log.[5] It also receives checkpoints directly from the edge agent and stores them. The local recovery process then works in exactly the same way as the remote recovery process (per session), but utilizes local checkpoints and logs rather than waiting for client and server to send these.

CESSNA maintains these two recovery mechanisms side-by-side during normal operation. Upon recovery, if recovering to a physically close edge, local recovery is used. Otherwise, remote recovery is selected. To achieve even faster recovery, CESSNA provides an optional hot backup mechanism in which a designated alternate edge is running adjacent to the active edge. The alternate edge does not process any incoming messages, but is updated with every new checkpoint that is taken. In case of a failure, the alternate edge is ready to immediately fetch the relevant message and interleave logs from the local recovery service and then execute the recovery algorithm, saving the time it takes to start a new edge application instance and to restore a checkpoint.

3.4 Formalizing Our Guarantees

The previous discussion of CESSNA's design has many moving parts, which obfuscates the properties it can ensure. Here we collect the various assumptions about our solution, and summarize which properties they collectively guarantee. Due to space constraints, we do not formally define and prove these guarantees here, but rather provide an outline through a short discussion.

An edge application is a state machine, which has an initial state and a transition function. The latter is merely the application logic that responds to *inputs*: messages (from the client and the server) and events (*e.g.,* thread scheduling and time based decisions). Our correctness guarantee, which we call *output message consistency*, translates to guaranteeing that upon recovery, CESSNA reinstates a state machine with the same initial state and transition function (*i.e.,* same application), positioned at the last committed state of the original

[5] We assume that if TLS is used, it is terminated before the ToR switch of the edge application, as done by Google [40] and others. ToR tap port access is a requirement for using CESSNA's local recovery option though, as noted previously, other solutions could be used for local recovery.

state machine before it failed. We define the last committed state as one where transitioning to the state produced a message that was successfully received by the client or serve.

CESSNA provides output message consistency by reconstructing the lineage of messages and events at the client and the server. We piggyback updates to the interleave log on *every* message sent to client and server. Thus we are guaranteed that at least one of them has an up-to-date interleave log, which can be used to recover state starting from the initial system state. When recovering from a checkpoint, we merely replay the interleave log starting from an intermediate point to arrive at the last committed state.

3.5 Implementation

In order to evaluate the CESSNA design, we implemented two prototypes: each of our prototypes targets a different runtime engine.

The first, which we refer to as Container Isolated CESSNA (§3.5.1) is built on top of Docker and relies on Docker's checkpointing mechanism [16]. The use of Docker containers simplifies the adoption of CESSNA since the only code changes required are at the edge, where the changes are needed in order to enable message replay.

However, this ease of adoption comes at the cost of checkpoint overheads: Docker's checkpointing mechanisms are agnostic to application semantics, and thus container checkpoints include not just application data but also local state from the stack and other information which is unnecessary for replay.

The second, which we call Software Isolated CESSNA (§3.5.2), is built to provide checkpointable data structures. Programs need to be modified in order to use these checkpointable data structures, but this reduces checkpoint overheads since developers explicitly mark out what data is semantically essential for recovery, and Software Isolated CESSNA does not checkpoint any additional data.

We implemented both approaches in order to show that (a) CESSNA can be employed by existing edge applications with minimal changes in order to achieve fault tolerance; and (b) the cost of fault tolerance can be made negligible for future CESSNA-aware edge applications. We describe both implementations below.

3.5.1 Container Isolated CESSNA (CI-CESSNA)

CI-CESSNA is our transparent implementation that uses Docker, and can provide fault tolerance for any application that uses TCP for communication between client, edge, and server. The client and server code can transparently use CI-CESSNA via the socket interposition layer and connection handlers (described below). For the edge code, we require the application to be written using CESSNA's Edge API in order enable message replay, however no changes are required to the application logic.

Below we describe the client, edge, and server components that comprise CI-CESSNA.

Client/Server Components

Client Socket Interposition Layer: The socket interposition layer is used to allow unmodified client applications to use CESSNA transparently. It is a small piece of C++ code that interposes on socket `connect()` calls. If the call is associated with a CESSNA application, a new session is created and the interposed code connects to the corresponding local connection handler.

The interposition layer is a shared library that is loaded dynamically using the `LD_PRELOAD` environment variable. This enables applications written in any language to use the library with no modification. Only CESSNA applications need to preload the interposition layer, so it does not affect other applications on the client machine.

Host Agent: The host agent is responsible for receiving checkpoints and for managing session life-cycles. The host agent communicates with its corresponding edge agent out-of-band, in parallel to the application session using a REST API over HTTP. Messages are encoded with JSON. In each host, the host agent is also responsible for starting a *connection handler* for each session.

Connection Handler: Connection handlers are implemented as TCP proxies, which implement the CESSNA data plane protocol and provide the host agents with the outgoing message logs and interleave logs extracted from incoming messages. In a client host, the client application connects to the TCP proxy, and the TCP proxy connects to the edge on behalf of the client. In a server host, the TCP proxy accepts connections from the edge on behalf of the server, and the TCP proxy connects to the server.

Edge Components

Edge Agent: The edge agent manages checkpoints and communications with the host agents. The edge agent may run on a different physical machine than the runtime engine, and can manage multiple Docker engines on multiple physical machines. Upon receiving a new session request, the edge agent forwards it to the corresponding server and waits for a response. When a response arrives, it spins up a container that runs the application's edge code.

Checkpoint and Recovery: The edge agent is responsible for taking checkpoints when requested by an application. To checkpoint state we use Docker's `checkpoint create` command which pauses the container, takes a checkpoint, and then resumes the container. To restore the checkpoint, we use Docker's `start` command with the checkpoint flag and ID. We measure the latency associated with these processes in Section 3.6. The checkpoint files are compressed and, depending on configuration, sent to the required remote destination(s).

Using CI-CESSNA

Edge Application API: We provide a CESSNA edge library to applications which implements the Edge API described in Section 3.3.1. The recovery process is also handled by the Edge API in case the application starts in a recovery mode. The edge library also provides additional methods for managing an application's life-cycle (*e.g.,* initialization, shutdown). Programmers can create a new edge application by subclassing the CESSNA application class (provided by the edge library) and overriding methods for handling edge events (*e.g.,* `recv_client_msg` which is invoked when a message is received).

3.5.2 Software Isolated CESSNA (SI-CESSNA)

In order to minimize checkpoint size we implemented a library of checkpointable data structures and communication primitives. Our original implementation was in Rust, and used gRPC for communication between client, edge and server. Subsequently, in order to apply CESSNA's methods to the Rocket Video Analytics Platform [4], we used the same techniques to implement CESSNA in C#. Since Rocket directly makes use of TCP sockets, our C# implementation also uses TCP sockets instead of gRPC or other RPC libraries.

In order to use the Rust implementation of SI-CESSNA, the client and server must be written using the Host API provided by CESSNA, but no application logic changes are required. For the C# implementation of SI-CESSNA, the client and server can transparently use CESSNA using the socket interposition layer and connection handlers as described above. When using SI-CESSNA (both Rust and C#), the edge must be written using CESSNA's Edge API, and requires the use CESSNA's checkpointable data structures.

Client/Server Components

Host API: For the Rust implementation, we provide a Host API for the client and server that allows them to establish a session and send and receive gRPC messages. For the C# implementation, we can use the socket interposition layer as described in section 3.5.1. The host API also includes modules to provide the functionalities of the host agent as described in Section 3.3.2.

Edge Components

API for improved checkpoint and recovery: The edge library provides a set of data structures for which we can compute checkpoints. We checkpoint the application state by serializing and saving the contents of these data structures, and restore them by deserializing the stored checkpoints. Edge applications then must use these data structures to store any state that persists across messages.

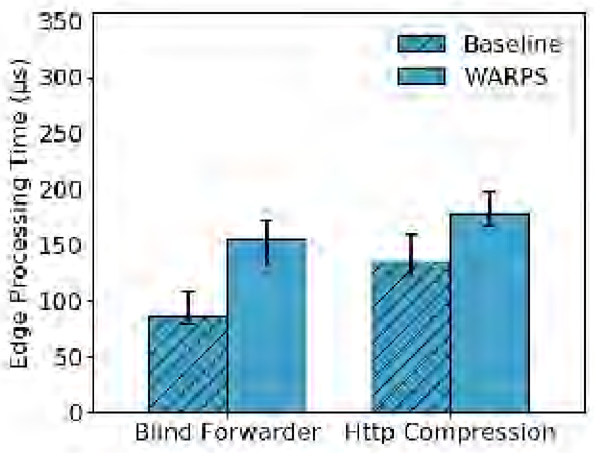


Figure 3.2: Median overhead of applications with CI-CESSNA, error bars drawn at 5th and 95th percentile latencies.

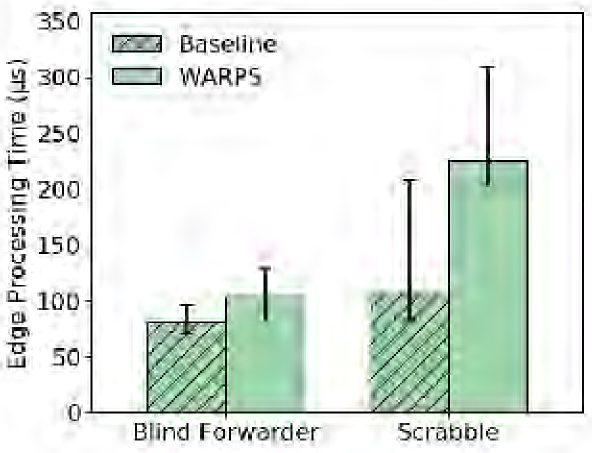


Figure 3.3: Median overhead of applications with SI-CESSNA, error bars drawn at 5th and 95th percentile latencies.

Edge Application API: We provide applications the same API as described in Section 3.3.1. In addition, the API provides the interface for creating and using checkpointable data structures, which are periodially checkpointed.

The recovery process is also handled by the Edge API in case the application starts in a recovery mode.

3.6 Evaluation

We evaluate CESSNA by addressing two questions, which we discuss in the sections below: (i) what are the overheads imposed by CESSNA in the absence of failures, and (ii) how long does it take to recover when an edge fails. To answer these questions, we developed four new applications and ported three existing applications to make use of CESSNA, which we now describe.

3.6.1 Applications on CESSNA

We implemented some sample applications atop CI-CESSNA (denoted by *) and some atop SI-CESSNA (denoted by †). For comparison, we also create a baseline version of each application, which runs without CESSNA's recovery functionality.

The applications we developed are:

Blind Forwarder *†**:** A simple edge application that forwards every message it receives to the other side of the edge. This is not a meaningful edge application, but it allows us to easily analyze the impact/overhead of CESSNA.

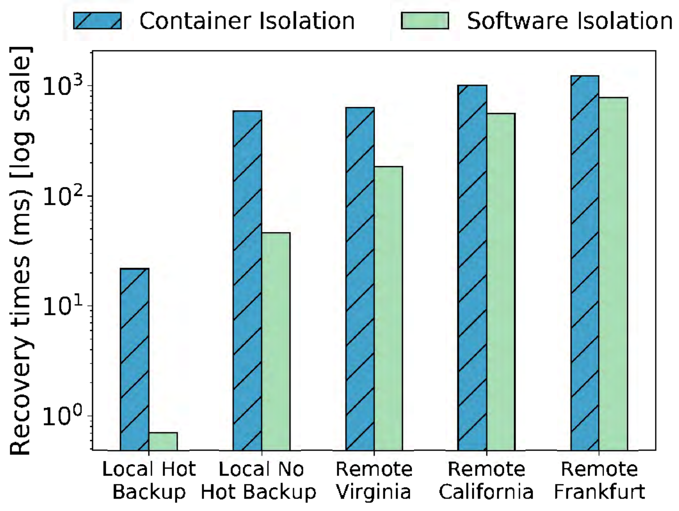


Figure 3.4: Latency overhead of the recovery process when the client, edge, and server are all in Virginia and remote recovery is as noted.

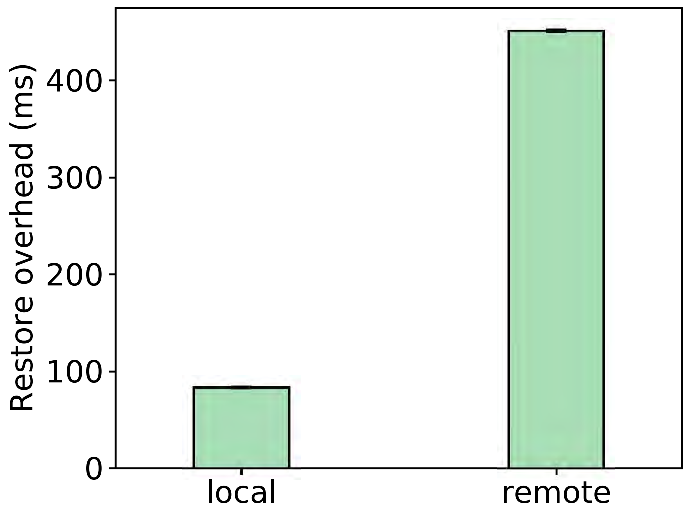
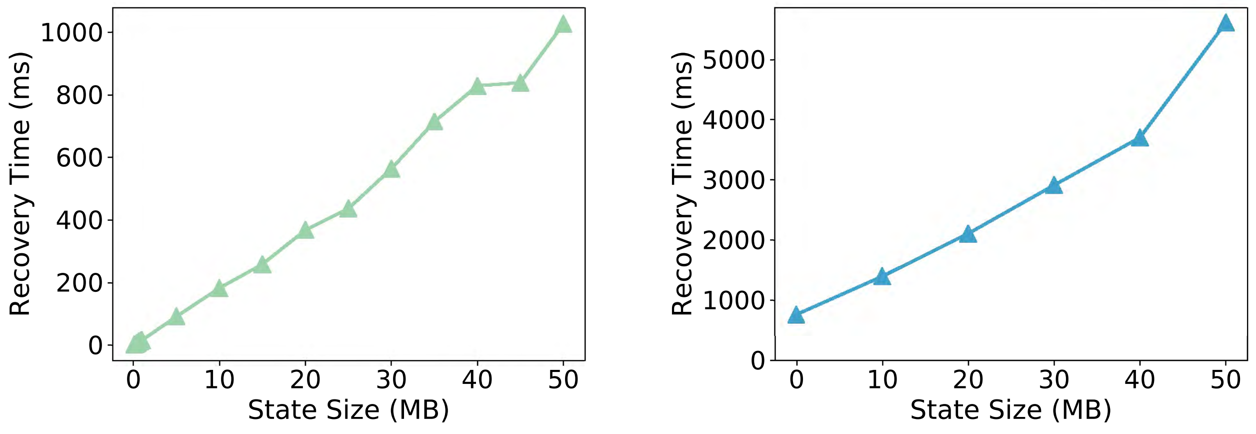


Figure 3.5: Median time taken to restore background subtraction state locally and remotely, error bars drawn at 5th and 95th percentile latencies.

Multi-Player Games: We wrote Battleship* and Scrabble† games from scratch to use the edge to provide fast responses to user actions and to offload user-related state and computation from the server. Specifically, the edge verifies user actions (*e.g.,* did the user chose valid words in Scrabble, or did they hit or miss a ship in Battleship), and synchronizes the game state with the server. In these applications, the edge reduces the response latency to the clients. For instance, in our setting (which we describe later), when submitting a



(a) SI-CESSNA (b) CI-CESSNA

Figure 3.6: Recovery time as a function of state size.

Scrabble move, the client experienced a median response latency of 414 µs with an edge, compared to 75.9 ms without an edge.

Existing Games: We modified for two *existing* open-source multiplayer games, Pong† and Snake*, by adding a stateful edge component. All rule checking and game object rendering is done at the edge, improving latency and reducing computation at the client. The main difference between these two games and the previous ones is that these are more similar to real-life multiplayer games: players are constantly in motion, and hundreds of messages are sent between the client, the edge, and the server each second.

Stateful Compression *: This edge application offloads compression from clients. Data is sent uncompressed between the client and the edge, and compressed between the edge and the server. We also extended this application to support de/compression of HTTP requests and streamed, chunked responses (similarly to [49]), and tested it with an unmodified Apache Tomcat 7 web server and an unmodified web-browser.

Video Analytics Application †: We extended the Microsoft Rocket Video Analytics Platform [4] in order to incorporate fault-tolerance. The Rocket platform is an extensible software stack meant for live analysis of video streams (*e.g.,* traffic cameras). The platform consists of a pipeline where the decoded frame is first passed through the OpenCV background subtraction module, then processed by a chain of Deep Neural Networks (DNNs). Prior work [29] has shown that running background subtraction and the first few DNNs at the edge can significantly reduce the amount of data forwarded to the cloud, by allowing the pipeline to filter out frames with no actionable information and by reducing the number of pixels contained in each frame. In order to do the latter, the background subtraction module must compute a scene background, which it does by averaging 120 frames spread over two minutes of video. Edge failure can result in a loss of this computed background, and during recovery the framework must either transfer additional data to the cloud (thus increasing network requirements) or pause analysis while the background is recomputed. Since the DNNs used by Rocket assume that inputs have already gone through an initial background subtraction step, Rocket currently pauses analysis while the background is recomputed, and this results in user-visible interruption. The other modules in this pipeline, including the DNNs, are all stateless.

In order to enable fault tolerance for Rocket, we modified the background subtraction module (specifically the OpenCV and OpenCVSharp libraries) to use SI-CESSNA data structures for storing the background and intermediate frames which are periodically averaged in order to compute a new background. We then use CESSNA to checkpoint this state, and restore it during failover, thus providing fault tolerance.

While our Rocket edge makes use of CESSNA-SI's checkpointable data structures, the current version does not make use of message replay from the clients (the cameras) because Rocket assumes the cameras have no computational capability (and thus have no ability to store the log and send it to the edge during recovery). While this precludes *perfect* state recovery, the partial state recovery from the checkpoints is sufficient to minimize disruption to the analytics pipeline. This limitation can be easily addressed in the future by the use of smart cameras.

This set of applications gave us a wide range of test cases. In the measurements to follow, for brevity we only discuss one or two of them for each question, but we measured the performance of all of them and the results presented here are representative of the broader set.

3.6.2 Performance Evaluation

We evaluated the performance of CESSNA by deploying it on multiple machines (of type `m5.xlarge`) in the AWS network, in different geographic locations.

What is the overhead in normal operation?

We first look at the overheads imposed by CESSNA in handling packets (where it must extract the message and interleave logs from the header and append to local copies). Figure 3.2 shows the median edge processing latency for blind forwarding and for HTTP compression implemented on CI-CESSNA. CESSNA's overhead is well below 100 µs. Figure 3.3 shows the median edge processing latency for Scrabble implemented on SI-CESSNA. When using SI-CESSNA, the results include time for generating checkpoints in addition to time for extracting message and interleave logs. As a result, SI-CESSNA adds 120 µs of edge processing time, due to the persisting of board state updates to disk after receiving every message (discussed below). These per-packet processing delays are minimal compared to typical round-trip times. The overhead added at the client and the server due to CESSNA's encapsulation is negligible.

Checkpointing is a more complicated story. Ideally, a checkpoint would be taken when the application begins and then incrementally updated as each message arrives. This is how we implemented checkpointing in SI-CESSNA, and the 120 µs overhead reported above is due to this incremental updating. However, Docker does not support incremental updating, so CI-CESSNA must periodically take full checkpoints. Assuming the checkpoint size is 10MB, this requires freezing packet processing for roughly 360 ms which is a sizable delay (suggesting that checkpoints should be taken infrequently). However, incremental checkpointing is widely available in other container and VM orchestration systems, and as reported on in [56] this can reduce pause intervals to a low 2–4 ms.

We next look at the overhead imposed by SI-CESSNA on the video analytics application due to checkpointing. For this experiment, we process a video of frame size 768×576 with a total of 795 frames, which requires checkpointing 43MB of state.

The median overhead of taking a checkpoint is 121 ms. However, scene backgrounds – which are found by computing the moving average of input video frames over time – change slowly over the course of several seconds or even minutes. As a result one need not checkpoint at the granularity of a single frame, and checkpointing once every second or even minute suffices, thus reducing checkpointing overhead. The time to checkpoint increases linearly as the checkpoint state size grows.

In terms of bandwidth overhead, our data plane protocol adds between 12 bytes (host $\rightarrow$ edge messages) to 36 bytes (edge $\rightarrow$ host) to each message. It is currently optimized for simplicity and not size, but even this straightforward version adds less than 2% overhead to 1500 byte packets. Messages on the control plane are rare and short except for checkpoints (where size is dependent on the application, and on whether they are incremental or not).

How long does it take to recover?

Figure 3.4 shows the latency overhead of the recovery process, for both local and remote recovery. In order to test this, we used a version of the blind forwarder which stored arbitrary state of a given size so we can control the size of the snapshot. We crashed the active edge and measured additional latency during recovery.

CI-CESSNA: Local recovery with a hot backup incurs a latency overhead of 21 ms (median result), which is mostly due to our recovery algorithm (Algorithm 1). Local recovery of a 10MB checkpoint without a hot backup incurs 585 ms overhead. The additional overhead is mainly due to Docker's checkpoint restore command (68 %), while the CI-CESSNA agent incurs another 27 % for preparing the checkpoint, decompressing it, and verifying the recovered container before resuming the session. Remote recovery also adds link latencies.

SI-CESSNA: There is a substantial improvement in recovery using SI-CESSNA: the latency overhead for local recovery with a hot backup is 0.71 ms (median result), while the overhead for local recovery without a hot backup is 46 ms when restoring a 10MB checkpoint. Remote recovery in the same AWS region has 183 ms latency overhead. The replay process in this experiment replayed 50 messages. Replaying more messages would have linearly increased the overhead, at a rate of about 10s of µs per replayed message; for reasonable numbers of messages this would add very little additional delay.

For the video analytics application, our experiment setup consisted of two machines (each with an Intel Xeon CPU E5-2660 v3, 128GB RAM) in the same rack. Figure 3.5 shows the time it would take to (a) restore a checkpoint stored locally (from a file), and (b) transmit and restore a checkpoint at a nearby edge connected via a 1Gbps link with a measured min RTT of 116 µs. The median overhead to recover locally is 83 ms and the median overhead to recover remotely is 450 ms. A major component in remote recovery time is the transmission of the checkpoint (the theoretical transmission time of the 43MB checkpoint at 1Gbps is 340 ms), with the restore process (from memory) only taking around a median of 62 ms. The time to transmit a checkpoint can be reduced by either compressing checkpoints or using faster links.

Figures 3.6a and 3.6b show the time for local recovery without a hot backup as a function of the checkpoint size when using SI-CESSNA and CI-CESSNA respectively. Recovery time for both grows linearly with increase in checkpoint size. In our approach, checkpoints only capture state associated with a single session, so we expect that checkpoints could be small in many cases, leading to reasonable recovery times.

3.6.3 Summary and Discussion

Our results suggest that CESSNA can support a wide range of applications, that stateful processing can be beneficial, and that the performance overheads can be reasonable. For instance, SI-CESSNA has low packet processing overheads (even while continuously taking checkpoints), and can recover from failure in less than 1 ms with a local hot standby, and in less than 50 ms for a local cold standby. However, SI-CESSNA requires applications to be written in a supported language (at present C# or Rust), which may hinder adoption.

CI-CESSNA is easier to adopt as it imposes no limitation on the choice of application language for the client and the server (the edge API for CI-CESSNA is currently provided only in Python). The packet processing latencies remain low, but the checkpoint delays can be substantial, and in turn the recovery delays are similarly inflated by Docker's slow processing of checkpoints. However, if failures are infrequent, these delays might be tolerable.

Deployment and Scalability Since CESSNA is based on sessions, it handles each client-edge-server session independently of other sessions. Therefore, it is relatively simple to deploy and scale. Standard load balancing techniques can be used to select an edge machine given a new session request. A single edge agent can manage edge sessions on multiple physical machines. Moreover, if migration of an existing session is needed, the process is inherently supported as the existing edge process can be killed and a new one will automatically recover and continue to serve the application (with some transient delay due to the recovery process as described above).

3.7 Conclusion

While strongly stateful edge computation is already in use, its correctness and reasonable performance under failure and mobility is typically not guaranteed by current approaches. We propose a framework, applicable to any session-oriented application whose edge obeys our requirements from clients and servers, that provides correctness and reasonable performance for such applications. Moreover, we provide two reference implementations for our proposed design: one shows that our design can be easily deployed using industry standard runtime engines, but introduces some (reasonable) overheads; the other shows that using an optimized API and runtime environment leads to significantly lower performance overheads. Both implementations demonstrate that message replay and checkpoint based mechanisms can be adopted to provide fault tolerance at the edge.

Bibliography

[1] Marcos K Aguilera et al. "Microsecond consensus for microsecond applications". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 599–616.

[2] Ailidani Ailijiang et al. "WPaxos: Wide Area Network Flexible Consensus". In: *IEEE Transactions on Parallel and Distributed Systems* 31 (2019), pp. 211–223.

[3] *Akamai: Cloudlet Applications*. `https://www.akamai.com/us/en/products/performance/cloudlets/`.

[4] Ganesh Ananthanarayanan et al. *Project Rocket platform—designed for easy, customizable live video analytics—is open source*. Microsoft Research Blog. Jan. 2020. URL: `https://www.microsoft.com/en-us/research/publication/project-rocket-platform-designed-for-easy-customizable-live-video-analytics-is-open-source/`.

[5] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. "The datacenter as a computer: Designing warehouse-scale machines". In: *Synthesis Lectures on Computer Architecture* 13.3 (2018), pp. i–189.

[6] Kenneth P. Birman and Thomas A. Joseph. "Exploiting virtual synchrony in distributed systems". In: *SOSP '87*. 1987.

[7] Luiz Fernando Bittencourt et al. "Towards Virtual Machine Migration in Fog Computing". In: *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)* (2015).

[8] Anita Borg, Jim Baumbach, and Sam Glazer. "A message system supporting fault tolerance". In: *SOSP '83*. 1983.

[9] Broadcom. *Broadcom Trident4/BCM56880 Series*. `https://www.broadcom.com/products/ethernet-connectivity/switching/strataxgs/bcm56880-series`. 2020.

[10] Mike Burrows. "The Chubby lock service for loosely-coupled distributed systems". In: *OSDI*. 2006.

[11] Tushar Deepak Chandra and Sam Toueg. "Unreliable failure detectors for reliable distributed systems". In: *J. ACM* 43 (1996), pp. 225–267.

[12] James C. Corbett et al. "Spanner: Google's Globally-Distributed Database". In: *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. Hollywood, CA: USENIX Association, Oct. 2012, pp. 261–264. ISBN: 978-1-931971-96-6. URL: `https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett`.

[13] Natacha Crooks et al. "Seeing is Believing: A Client-Centric Specification of Database Isolation". In: *PODC*. 2017, pp. 73–82.

[14] Brendan Cully et al. "Remus: High Availability via Asynchronous Virtual Machine Replication". In: *NSDI*. 2008.

[15] Huynh Tu Dang et al. "P4xos: Consensus as a network service". In: *IEEE/ACM Transactions on Networking* 28.4 (2020), pp. 1726–1738.

[16] *Docker Checkpoint and Restore*. `https://github.com/docker/cli/blob/master/experimental/checkpoint-restore.md`. 2018.

[17] George W. Dunlap et al. "Execution Replay of Multiprocessor Virtual Machines". In: *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2008, pp. 121–130.

[18] George W. Dunlap et al. "ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay". In: *ACM SIGOPS Oper. Syst. Rev.* 36.SI (Dec. 2003), pp. 211–224.

[19] *Durable Functions Overview*. `https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview`. 2019.

[20] Vitor Enes et al. "Efficient Synchronization of State-Based CRDTs". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), pp. 148–159.

[21] *etcd*. `https://coreos.com/etcd/`.

[22] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. "Impossibility of distributed consensus with one faulty process". In: *Journal of the ACM (JACM)* 32.2 (1985), pp. 374–382.

[23] Michael J. Freedman, Karthik Lakshminarayanan, and David Mazières. "OASIS: Anycast for Any Service". In: *NSDI*. 2006.

[24] Jonathan Goldstein et al. "A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications". In: *Proc. VLDB Endow.* 13.5 (2020), pp. 588–601.

[25] Rachid Guerraoui et al. "Can 100 machines agree?" In: *arXiv preprint arXiv:1911.07966* (2019).

[26] Rachid Guerraoui et al. "Throughput optimal total order broadcast for cluster environments". In: *ACM Transactions on Computer Systems (TOCS)* 28.2 (2010), pp. 1–32.

[27] Kiryong Ha et al. *Adaptive VM Handoff Across Cloudlets*. Tech. rep. Carnegie Mellon University, 2015.

[28] Yotam Harchol et al. "Making edge-computing resilient". In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 253–266.

[29] Kevin Hsieh et al. "Focus: Querying Large Video Datasets with Low Latency and Low Cost". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018.

[30] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *ATC*. 2010.

[31] Intel Corporation. *Intel Tofino*. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html`. 2020.

[32] Sagar Jha et al. "Derecho: Fast state machine replication for cloud services". In: *ACM Transactions on Computer Systems (TOCS)* 36.2 (2019), pp. 1–49.

[33] Xin Jin et al. "NetChain:Scale-Free Sub-RTT Coordination". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 2018, pp. 35–49.

[34] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. "ZAB: High-performance broadcast for primary-backup systems". In: *DSN*. 2011.

[35] Marios Kogias and Edouard Bugnion. "HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: `10.1145/3342195.3387545`. URL: `https://doi.org/10.1145/3342195.3387545`.

[36] Jonathan Corbet. *Checkpoint/Restart in Userspace*. LWN `https://lwn.net/Articles/572125/`. 2013.

[37] Leslie Lamport. "Fast paxos". In: *Distributed Computing* 19.2 (2006), pp. 79–103.

[38] Leslie Lamport. *Generalized Consensus and Paxos*. Tech. rep. MSR-TR-2005-33. Microsoft Research, 2005.

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

[40] Adam Langley et al. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: *SIGCOMM*. 2017, pp. 183–196.

[41] L Leslie. "The part-time parliament". In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169.

[42] Jialin Li et al. "Just say NO to paxos overhead: Replacing consensus with network ordering". In: *OSDI*. 2016.

[43] Barbara Liskov and James Cowling. *Viewstamped Replication Revisited*. Tech. rep. MIT-CSAIL-TR-2012-021. MIT, July 2012.

[44] Yanhua Mao, Flavio P Junqueira, and Keith Marzullo. "Mencius: building efficient replicated state machines for WANs". In: *OSDI*. 2008.

[45] Parisa Jalili Marandi et al. "Ring Paxos: A high-throughput atomic broadcast protocol". In: *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2010, pp. 527–536.

[46] Christopher Meiklejohn and Peter Van Roy. "Lasp: a language for distributed, coordination-free programming". In: *PPDP*. 2015.

[47] Iulian Moraru, David G Andersen, and Michael Kaminsky. "There is more consensus in egalitarian parliaments". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 358–372.

[48] *Multi-access Edge Computing (MEC); Phase 2: Use Cases and Requirements*. `https://www.etsi.org/deliver/etsi_gs/MEC/001_099/002/02.01.01_60/gs_MEC002v020101p.pdf`. 2018.

[49] NGINX Inc. *Compression and Decompression*. `https://docs.nginx.com/nginx/admin-guide/web-server/compression/`. 2019.

[50] Edmund B. Nightingale et al. "Rethink the Sync". In: *OSDI*. 2006, pp. 1–14.

[51] Brian M Oki and Barbara H Liskov. "Viewstamped replication: A new primary copy method to support highly-available distributed systems". In: *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*. 1988, pp. 8–17.

[52] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *USENIX ATC*. 2014.

[53] Haochen Pan et al. "Rabia: Simplifying State-Machine Replication Through Randomization". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 472–487.

[54] Dan RK Ports et al. "Designing distributed systems using approximate synchrony in data center networks". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 43–57.

[55] *Home Security Systems — Smart Home Automation — Ring*. `https://ring.com/`. 2020.

[56] Adam Ruprecht et al. "VM Live Migration At Scale". In: *VEE*. 2018.

[57] Mahadev Satyanarayanan. "The Emergence of Edge Computing". In: *IEEE Computer* 50.1 (2017), pp. 30–39.

[58] Mahadev Satyanarayanan et al. "The Case for VM-Based Cloudlets in Mobile Computing". In: *IEEE Pervasive Computing* 8.4 (2009), pp. 14–23.

[59] Daniel J. Scales, Mike Nelson, and Ganesh Venkitachalam. “The design of a practical system for fault-tolerant virtual machines”. In: *Operating Systems Review* 44 (2010), pp. 30–39.

[60] Fred B Schneider. “Implementing fault-tolerant services using the state machine approach: A tutorial”. In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319.

[61] Marc Shapiro et al. “Conflict-Free Replicated Data Types”. In: *SSS*. 2011.

[62] Justine Sherry et al. “Rollback-Recovery for Middleboxes”. In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*. 2015.

[63] Jan Skrzypczak, Florian Schintke, and Thorsten Schütt. “Linearizable State Machine Replication of State-Based CRDTs without Logs”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019).

[64] Michael Stonebraker et al. “The End of an Architectural Era: (It’s Time for a Complete Rewrite)”. In: VLDB ’07. Vienna, Austria: VLDB Endowment, 2007, pp. 1150–1160. ISBN: 9781595936493.

[65] Jeff Terrace and Michael J Freedman. “Object storage on CRAQ: High-throughput chain replication for read-mostly workloads”. In: *USENIX Annual Technical Conference*. 2009.

[66] Douglas B Terry et al. “Managing update conflicts in Bayou, a weakly connected replicated storage system”. In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 172–182.

[67] Robbert Van Renesse and Deniz Altinbuken. “Paxos made moderately complex”. In: *ACM Computing Surveys (CSUR)* 47.3 (2015), pp. 1–36.

[68] Robbert Van Renesse and Fred B Schneider. “Chain Replication for Supporting High Throughput and Availability.” In: *OSDI*. Vol. 4. 2004.

[69] Limin Wang, Vivek S. Pai, and Larry L. Peterson. “The Effectiveness of Request Redirection on CDN Robustness”. In: *OSDI*. 2002.

[70] *Wyze — Making Great Technology Accessible — Smart Home Devices*. `https://wyze.com/`. 2020.

[71] Zhuolong Yu et al. “Netlock: Fast, centralized lock management using programmable switches”. In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 126–138.

[72] Irene Zhang et al. “Optimizing VM Checkpointing for Restore Performance in VMware ESXi”. In: *USENIX Annual Technical Conference*. 2013.