Making Edge-Computing Resilient

Yotam Harchol
EPFL
yotamhc@cs.berkeley.edu

Aisha Mushtaq
UC Berkeley
aisha@berkeley.edu

Vivian Fang
UC Berkeley
v.fang@berkeley.edu

James McCauley
UC Berkeley, ICSI
murphy@berkeley.edu

Aurojit Panda
NYU
apanda@cs.nyu.edu

Scott Shenker
UC Berkeley, ICSI
shenker@icsi.berkeley.edu

Abstract

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 this paper we propose a design for meeting this challenge called the Client-Edge-Server for Stateful Network Applications (CESSNA).

CCS Concepts

• **Networks** → *Programming interfaces*; • **Computer systems organization** → **Fault-tolerant network topologies**.

Keywords

Edge Computing, Fault Tolerance

ACM Reference Format:
Yotam Harchol, Aisha Mushtaq, Vivian Fang, James McCauley, Aurojit Panda, and Scott Shenker. 2020. Making Edge-Computing Resilient. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3419111.3421278

1 Introduction

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 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.

[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.

SoCC '20, October 19–21, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8137-6/20/10.
https://doi.org/10.1145/3419111.3421278

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.

2 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.

2.1 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 [29, 30] 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 [2]. 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 [5, 18] 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 [14] 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 [19, 23, 27, 38].

2.2 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 [32] and closely related work such as viewstamped replication [26] and virtual synchrony [4] 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 [7] 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 [6], Remus [10], and other systems have relied on VM and process replication [31] 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 [12], SMP-ReVIRT [13], and FTMB [34] 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 [18] 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 [1] and Mencius [21] 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 [15, 22, 35] have built on conflict-free replicated data types [33] (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), 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 4.

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 [17], Bayou [36], and the proposal by Nightingale et al [25]; and reformulations of existing models [9]. Consistency models in distributed data stores dictate when *updates are visible* to different clients. While we similarly describe a consistency model in the paper, 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 Our Approach

3.1 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 paper 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.

3.2 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).

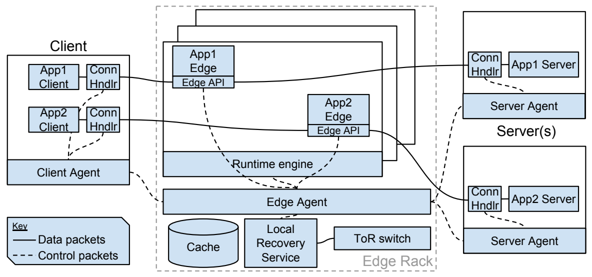


Figure 1: The general design of our framework.

4 CESSNA's Design

In this section we describe the design of CESSNA (illustrated in Figure 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.

4.1 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 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 Section 5, 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 [11], KVM [8], and VMware [39]. 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.

[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 1: Methods provided by the Edge Application API.

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 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.

[4]We return to the issue of checkpoints in Section 7, where we note that checkpoints can be incrementally computed, greatly reducing the time the framework needs to wait.

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 [29]), the cache fetches missing items from the server, so every read operation to the cache returns a result.

4.2 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 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 [16, 37] can be used for this purpose.

4.3 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 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

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 check-
    point_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 in-
    terleave 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
```

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.

5 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 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.

[5]We assume that if TLS is used, it is terminated before the ToR switch of the edge application, as done by Google [20] 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.

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.

6 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 (§6.1) is built on top of Docker and relies on Docker's checkpointing mechanism [11]. 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 (§6.2), 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.

6.1 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.

6.1.1 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.

6.1.2 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 7. The checkpoint files are compressed and, depending on configuration, sent to the required remote destination(s).

6.1.3 Using CI-CESSNA

Edge Application API: We provide a CESSNA edge library to applications which implements the Edge API described in Section 4.1. 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).

6.2 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 [3], 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.

6.2.1 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 6.1.1. The host API also includes modules to provide the functionalities of the host agent as described in Section 4.2.

6.2.2 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.

Edge Application API: We provide applications the same API as described in Section 4.1. 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.

7 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.

7.1 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.

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 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

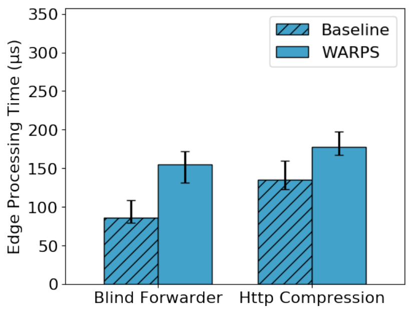


Figure 2: Median overhead of applications with CI-CESSNA, error bars drawn at 5th and 95th percentile latencies.

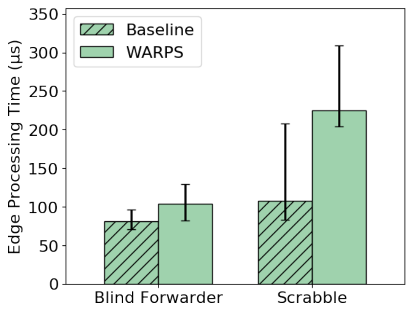


Figure 3: Median overhead of applications with SI-CESSNA, error bars drawn at 5th and 95th percentile latencies.

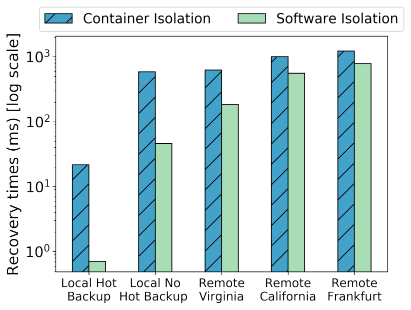


Figure 4: Latency overhead of the recovery process when the client, edge, and server are all in Virginia and remote recovery is as noted.

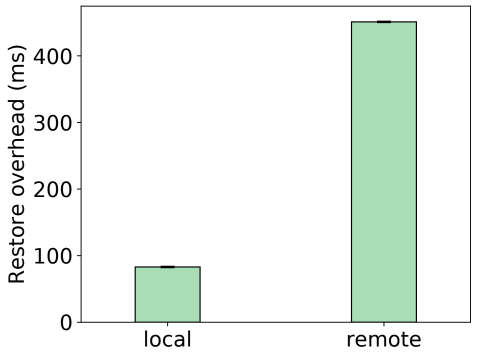
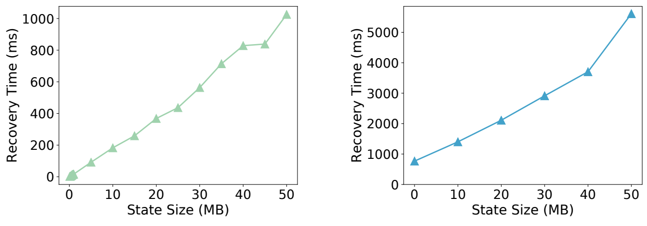


Figure 5: Median time taken to restore background subtraction state locally and remotely, error bars drawn at 5th and 95th percentile latencies.



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

Figure 6: Recovery time as a function of state size.

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 [24]), and tested it with an unmodified Apache Tomcat 7 web server and an unmodified webbrowser.

Video Analytics Application †: We extended the Microsoft Rocket Video Analytics Platform [3] 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 [19] 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.

7.2 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.

7.2.1 *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 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 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 [28] 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→edge messages) to 36 bytes (edge→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).

7.2.2 *How long does it take to recover?*

Figure 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 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 6a and 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.

7.3 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).

8 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. This paper proposes 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.

Acknowledgements

We thank our shepherd Indranil Gupta, and the anonymous reviewers for their helpful comments. We thank Ganesh Ananthanarayanan for discussions that led to our incorporation of the video workload. This work was funded in part by grants from Microsoft, Intel, VMware, Ericsson, Futurewei, and Cisco, and from NSF grants 1704941, 1817115, and 1817116.

References

[1] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2019. WPaxos: Wide Area Network Flexible Consensus. *IEEE Transactions on Parallel and Distributed Systems* 31 (2019), 211–223.

[2] akamai [n.d.]. Akamai: Cloudlet Applications. https://www.akamai.com/us/en/products/performance/cloudlets/.

[3] Ganesh Ananthanarayanan, Yuanchao Shu, Landon Cox, and Victor Bahl. 2020. Project Rocket platform—designed for easy, customizable live video analytics—is open source. Microsoft Research Blog. https://www.microsoft.com/en-us/research/publication/project-rocket-platform-designed-for-easy-customizable-live-video-analytics-is-open-source/

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

[5] Luiz Fernando Bittencourt, Marcio Moraes Lopes, Ioan Petri, and Omer F. Rana. 2015. Towards Virtual Machine Migration in Fog Computing. *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)* (2015).

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

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

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

[9] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *PODC*. 73–82.

[10] Brendan Cully, Geoffrey Lefebvre, Dutch T. Meyer, Mike Feeley, Norman C. Hutchinson, and Andrew Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*.

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

[12] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2003. ReVirt: Enabling Intrusion Analysis through

Virtual-Machine Logging and Replay. *ACM SIGOPS Oper. Syst. Rev.* 36, SI (Dec. 2003), 211–224.

[13] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. 2008. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 121–130.

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

[15] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. 2019. Efficient Synchronization of State-Based CRDTs. *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), 148–159.

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

[17] Jonathan Goldstein, Ahmed S. Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Christopher Meiklejohn, Umar Farooq Minhas, Ryan Newton, Rahee Peshawaria, Tal Zaccai, and Irene Zhang. 2020. A.M.B.R.O.S.I.A: Providing Performant Virtual Resiliency for Distributed Applications. *Proc. VLDB Endow.* 13, 5 (2020), 588–601.

[18] Kiryong Ha, Yoshihisa Abe, Zhuo Chen, Wenlu Hu, Brandon Amos, Padmanabhan Pillai, and Mahadev Satyanarayanan. 2015. *Adaptive VM Handoff Across Cloudlets*. Technical Report. Carnegie Mellon University.

[19] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*.

[20] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *SIGCOMM*. 183–196.

[21] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machine for WANs. In *OSDI*.

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

[23] multi-access 2018. 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.

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

[25] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the Sync. In *OSDI*. 1–14.

[26] Brian M. Oki and Barbara H. Liskov. 1988. 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*.

[27] ring 2020. Home Security Systems | Smart Home Automation | Ring. https://ring.com/.

[28] Adam Ruprecht, Danny Jones, Dmitry Shiraev, Greg Harmon, Maya Spivak, Michael Krebs, Miche Baker-Harvey, and Tyler Sanderson. 2018. VM Live Migration At Scale. In *VEE*.

[29] Mahadev Satyanarayanan. 2017. The Emergence of Edge Computing. *IEEE Computer* 50, 1 (2017), 30–39.

[30] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres, and Nigel Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing* 8, 4 (2009), 14–23.

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

[32] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.* 22 (1990), 299–319.

[33] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *SSS*.

[34] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. 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*.

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

[36] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 172–182.

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

[38] wyze 2020. Wyze | Making Great Technology Accessible | Smart Home Devices. https://wyze.com/.

[39] Irene Zhang, Tyler Denniston, Yury Baskakov, and Alex Garthwaite. 2013. Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In *USENIX Annual Technical Conference*.