Making Edge-Computing Resilient

Yotam Harchol
Aisha Mushtaq
Vivian Fang
James McCauley
Aurojit Panda
Scott Shenker

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2019-9
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-9.html

April 28, 2019

Making Edge-Computing Resilient

Abstract

The introduction of computational resources at the network edge allows us to offload computation from clients and/or servers, thereby reducing response latency, backbone bandwidth, and computational requirements on clients. More fundamentally, edge-computing moves us from a client-server model to a client-edge-server model. While this is an attractive paradigm for many applications, 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 stateful. In this paper we propose a design for meeting this challenge called the Client-Edge-Server for Stateful Network Applications (CESSNA).

1 Introduction

The recent introduction of compute and storage resources at the network edge[1] allows service providers to offer lower latency and higher throughput by directing clients to geographically-nearby content and computation. While edge computing has long been used to provide content-oriented caching to support video streaming and CDNs, it is now being used for new application areas such as IoT (where the edge pre-processes sensor data to improve responsiveness and reduce upstream bandwidth) and online games (where the edge improves client latency and allows light-weight clients to offload computation).

Network applications have long been based on the client-server paradigm, where a stateful server (or set of servers) provides services to multiple clients. While the fault tolerance issues of the client-server paradigm have been thoroughly studied [27, 15, 3], their solutions do not handle the case where a stateful edge processor fails. Addressing this gap in the fault-tolerance literature is the focus of our paper.

This gap has not received much attention because most existing edge platforms either recommend or enforce that edge applications[2] be effectively stateless: Cloudlets [26] and CDNs restrict applications to keeping immutable copies of data from a backend server [25] (we further discuss these in Section 8), and serverless

[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]By *edge application* we mean application code running at the edge.

offerings such as AWS Lambda@Edge and Cloudflare Workers provide no inherent recovery mechanisms, and are intended for stateless processing [1]. Statelessness makes the edge platforms simpler, but it limits the benefits available to the applications.

Multiplayer gaming is an example of a type of application that can benefit from stateful edges. Gaming companies have been working hard to reduce response latency for their customers, while still having a centralized server synchronizing a large number of players [20]. A stateful edge could help improve latency of multiplayer games, as we also show in our experiments, without the need for complex custom solutions.

A stateful edge introduces a new client-edge-server paradigm. With the original client-server paradigm, *fate sharing* is assumed to exist between the client and the server, such that if one of them dies, the session dies. While there are multiple techniques that have been developed to improve server resilience (*e.g.,* using replicated state machines, etc.) and client resilience (*e.g.,* using multi-homing 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 stateful edge, the session's fate is now shared between all three entities. This is problematic since edge failure can terminate a session even when the client and server remain alive. Furthermore, many of the benefits of using an edge are derived from having a client communicate with the *geographically nearest* edge. Thus, session state stored at an edge complicates client mobility, requiring that a client either access the same edge for the entire duration of a session (leading to inefficiency) or that the edge be stateless (limiting utility).

In this work, we aim to alleviate these concerns, and provide a mechanism through which a session can survive both the failure of an edge, or a client migrating to a new edge. We believe any solution to this problem needs to satisfy the following four properties:

1. **Survivability**: Edge failure does not kill the session, as long as there exists an edge to fail over to.

2. **Correctness**: In case of client movement or an edge failure, the new edge (whether nearby the old edge or not) should send messages that are consistent with those already sent by the previous edge. We formally define this correctness property in Section 2.1.

3. **Transparency**: Client and server logic should not need to be changed to support edge recovery. In some cases the logic at the edge might need to be aware of the recovery mechanism.

4. **Performance**: The solution must not severely degrade the performance of the

application. Specifically, next generation edge frameworks aim at $5 - 20ms$ latency [2], so the solution's overhead must be orders of magnitude less than that.

Achieving correctness and survivability while maintaining performance is challenging when the edge application is stateful: edge applications may read and process messages from multiple sources (*e.g.,* client and server) simultaneously. Nondeterministic events such as thread scheduling, time-based operations, and random number generation also affect the application's state and should be recovered correctly.

We propose a framework for client-edge-server applications called CESSNA (Client-Edge-Server for Stateful Network Applications) that satisfies all these requirements. CESSNA is an edge runtime environment that is not coupled with any specific application logic and works with any client and server applications as long as they have been adapted to use an edge.

CESSNA uses a message replay mechanism designed specifically for the client-edge-server paradigm. Our message replay mechanism ensures that even if a session is moved to a different edge (no matter the reason), it preserves the correctness and transparency properties described above. Using message replay, we also ensure the survivability property is satisfied, as messages are logged with the client and the server and thus obey fate-sharing.

We have implemented two fully working prototypes of CESSNA, using two different sets of technologies: the first uses an off-the-shelf runtime platform (Docker) and provides an API in a common programming language (Python), which makes it easy to use. The second implementation is optimized for performance using our own runtime and API in the Rust language. We deployed these implementations in multiple locations worldwide, running several edge applications. We provide a formal proof for the correctness guarantee, and comprehensive experimental results to show that CESSNA provides these guarantees with minimal performance overhead.

2 Computational Model

The ability to run computation at the edge enables a range of functionality including: (i) offloading computation from the client (reducing computational and power requirements); (ii) offloading computation from the server (reducing resource requirements at the server and improving application response latency); and (iii) reducing the amount of data transferred between client and server by allowing data to be preprocessed, *e.g.,* compressed or aggregated. Thus, the edge can extend the power of the client, reach of the server, and reduce the cost/penalty of communication between them. How edge capabilities are utilized depend on the application, and as a result, one cannot think of the code deployed at the edge as merely a subset of the client code, or merely a subset of the server code. Code running at the edge might be formed by taking bits of both client and server functionality, or might even implement functionality implemented by neither.

Our design aims to impose no restrictions on how clients or servers are built, and in particular it does not preclude the use of application-level methods of recovering from client and/or server failures, or the use of replication or other techniques to increase the resiliency of the server or client. We do not explore these further because our focus is on preserving correctness after a change in the edge. Thus, in what follows we assume a single (logical) client and a single (logical) server.

We assume that 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 part of this session.

In our model, a session can either be terminated explicitly or implicitly (*i.e.,* due to client failure, server failure, or when no functional edge is reachable).

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. In particular, we require that any sources of non-determinism at the edge are instrumented by our framework and hence must be provided by it.

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 a single message (or packet) is processed using only one version of the state.

Servers We place no restrictions on the behavior of the server. Similar to the existing client-server paradigm, the server can service multiple clients and coordinate clients among each other.

Clients Like the servers, we place no restrictions on the behavior of the clients. We assume that clients can be mobile, and as a result they might connect to different edges over time.

2.1 Problem Statement

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*: a correctly recovered edge is expected, given the same sequence of messages

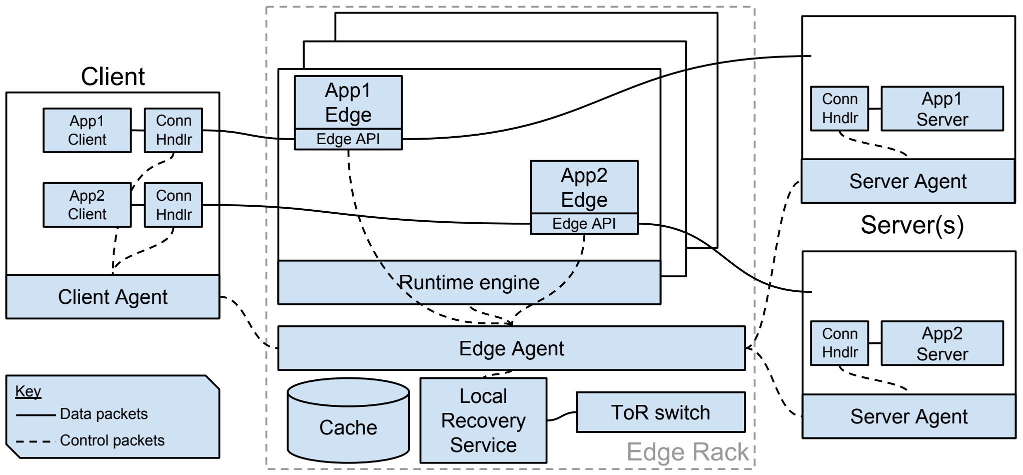

Figure 1: *The general design of our framework.*

and events, to emit exactly the same sequence of messages that would have been emitted by the original edge.

Output message consistency means that a recovered edge may not need to be recovered to the exact same state at which the failed edge stopped. Instead, it only needs to be recovered to the last *committed state* – the last time that the failed edge had emitted a message that was received by either the client or the server. This, to some extent, relaxes the recovery problem, and simplifies the solution for it: upon a failure, it is easier to recover a new edge to the last committed state than it is to the actual last processed state. We leverage this in our design.

3 CESSNA's Design

Here we present the design of CESSNA starting with some observations about client-edge-server applications that motivate our design.

3.1 Design Motivation

Upon movement of a host (a client or a server) to a new edge, we cannot assume that this edge has any state relevant to the session. Nor can we assume that CESSNA has any knowledge of the edge application logic (*i.e.,* we must treat the edge application largely as a black box). One way to recreate the state is to replay the messages seen by the previous edge. However, we have three observations about relying solely on the replay approach.

First, the edge receives messages from two different parties (the client and server), and these parties must keep a copy of the messages they sent to the edge at least until the edge can guarantee they are not needed anymore.

Second, the state of the edge may be dependent on the exact order in which these messages and any non-deterministic inputs and events (such as random number generation and thread scheduling) interleave. Thus, for any process of state reconstruction, we must have this ordering available, along with a record of non-deterministic events and the messages themselves. Since the edge can fail, this information must be stored elsewhere. One option is to send it to either the client or the server. Another option is to use some other storage (aside from client, server, or edge). We intermix the two options in our design as we describe later.

Third, storing all outgoing messages forever is impractical for many applications. However, we can use periodic snapshots in order to limit the size of the required message buffers, and reduce the time for session reconstruction.

Based on these observations our design accounts for two forms of recovery:

- **Local recovery** can be used when the failed edge and the new edge are physically close to each other, for example, in the same rack. Here there may be local storage of the relevant state.

- **Remote recovery** refers to the case when the old and new edges are far from each other. In this case, the client and the server cooperate with the newly provisioned edge to restore its state.

Our design, based on these observations, is illustrated in Figure 1, and we now discuss the design of each component in the figure.

3.2 Edge Design

The core of CESSNA is an edge runtime environment that allows seamless local and remote failover of edge applications.

3.2.1 Runtime Engine

Edge applications may run on top of any runtime engine that is able to take snapshots of running instances and to restore an instance given such a snapshot. For example, the runtime engine can be some VM/container technology, such as Docker [9], KVM [6], and VMware [32]. Our design is not tied to a particular runtime engine.

When using such runtime engines, edge applications have almost no restrictions, and in particular they are not limited to specific programming languages or uncommon libraries. However, these runtime engines are not optimized for fast snapshotting and restore, and as we show in Section 6, they introduce latency and degrade performance. Thus, we also propose a runtime engine optimized for fast snapshots and recovery, but which requires the use of a specific programming language and a specific set of libraries. This is described in Section 5.

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.*

3.2.2 Edge Agent

The edge agent 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.

3.2.3 Edge Application API

We designed an edge application API that provides the methods shown in Table 1, as well as a set of event handlers as described below.

The CESSNA edge framework needs to track the actual interleaving order in which messages from both the client and the server are processed by an edge application (see also Section 3.3). To do that, we provide the following event handler methods, which are to be overridden by the application: `accept_connection`, which is called when a client connects, and `recv_client_msg` and `recv_server_msg`, which are called when a message is received from the client or the server, respectively.

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 *interleave log*, which indicates the order in which messages were actually processed by the application, along with information on non-deterministic events, as described below.

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

In order to identify incoming and outgoing messages 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] When a message is received, the underlying edge framework reads its

[3]Header for host-¿edge messages is 12 bytes long and for edge-¿host is at least 36 bytes long (longer if the update for the interleave log is longer than one record). We note that this is optimized

sequence number and adds it to the interleave log. When a message is sent (using either the `send_msg_to_client` method or the `send_msg_to_server` method) it is sequenced and any update to the interleave log is attached to it.

3.2.4 Non-deterministic Operations

CESSNA edge applications can perform non-deterministic operations as long as the recovery process is deterministic. To allow this, our edge application API provides methods for non-deterministic operations and stores the order of their invocation and their returned values in the interleave log. Specifically, we introduce the `random` and `now` methods for random number generation and retrieving the current time respectively. During replay these do not generate new values and instead retrieve the original value from the interleave log of the corresponding thread which enables deterministic recovery using the algorithm shown in Section 3.3.

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. During replay, the provided function is invoked in the proper order without delay.

3.2.5 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, and it is guaranteed that every read operation to the cache returns a result. The cache is expected to fetch missing items from the server. Note that this is typical behavior for a cache at the edge (e.g., in CDNs and Cloudlets [25]). We assume that cached content does not change over time. The `cache_read` method of the edge application API can be used synchronously or asynchronously.

A mechanism similar to the one used for timers is employed for asynchronous cache reads.

3.2.6 Multithreading and Locks

CESSNA supports multithreaded edge applications while imposing a few limitations. Threads must 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

for simplicity and not size.

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), for which we provide an API. These locks log every acquire operation to the interleave log, and upon replay, the locks maintain the same order of acquisition. Special care is required to correctly track the order in which locks are being used by different threads as this may change upon a replay, and the edge application framework makes sure to reproduce the exact same order in such a case. 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.

When an edge application is multithreaded, the interleave log also stores the ID of the thread corresponding to each entry, and the client and the server keep track of which thread issued which message coming from the edge (this information is added to the CESSNA header). Upon replay, the recovery algorithm devises the order of outgoing messages based on this information, and blocks threads when necessary to produce the same ordering of output messages as originally.

3.3 Edge Recovery

In general, our recovery model assumes that the edge application's state is a function of messages received from both the client and the server, as well as non-deterministic events.

In order to recover a failed edge application we could theoretically restart it and replay all messages in the correct interleaving order (assuming for simplicity that no other events were logged). However, storing the entire log of messages, and their order, may be impractical.

Thus, we also use snapshots to capture the state of an application at regular intervals allowing for messages processed before this time to be purged.

As mentioned above, CESSNA has two recovery modes: local and remote. Both are built from the same building blocks (message replay, interleave log, and snapshots) and use the same algorithm. The key difference stems from the circumstances under which they are used. Local recovery is used when the failed edge and its replacement share some relatively fast storage (*e.g.,* when failing over to another machine in the same rack). Remote recovery is an extension to this approach which is used when the failed edge and its replacement are arbitrarily distant.

Both recovery modes can be enabled for an edge application, with local recovery being preferred when possible, and remote recovery ensuring survivability. It is the availability of both modes that allows CESSNA to satisfy the strict recovery requirements while providing high performance.

Upon receiving a request for session recovery, the edge agent first fetches the latest snapshot (if one exists) from whoever has it. This is based on the recovery model (local or remote) and in the remote case, on who has it stored (client/server/another edge). If a snapshot exists, the agent restores the snapshot. Otherwise, it just creates a fresh image or container of the application. It then starts the application in recovery mode. The application then asks the edge agent for the message replay and for the interleave log from both client and server. This information is provided to the edge agent either by the client and the server, or the agent fetches it from the local recovery service, as further explained in Section 3.3.1.

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
- lrbs=(src, seq) - last record before snapshot
- lcrbs=(src, seq) - last record before lrbs that appears in the
  interleave log known to client and server
- 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: ordering ← longest(client_order, server_order, lcrbs)
    // Take the longest interleave log provided by both the
    // client and the server, starting at lcrbs.
 3: ordering[thread_id] ← split(ordering)
    // Per-thread interleave log
 4: out_ordering ← merge(mrc, mrs)
    // Merge log of outgoing messages. Use this log to reorder
    // outgoing messages.
 5: threads_wait_evt.clear() // Let threads invoke API calls
 6: Let lrbs_idx ← index of lrbs in ordering[main_thread]
 7: for each idx in ordering[main_thread][lrbs_idx+1:] 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 re-
            sult.
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
```

The CESSNA edge framework executes the recovery algorithm described in Algorithm 1 on the application's main thread. Additional threads created by the application follow a similar process when invoking API methods during recovery. Special care is taken to ensure additional threads cannot invoke API calls before initialization has completed, even when the application is restored from a snapshot.

The core of the recovery process is message replay. The replay process has two purposes: first, to restore the state of the application, and second, to emit messages that would have been emitted had the application never failed. However, we would like to avoid re-emitting *all* messages, suppressing ones which we know must have been sent and received already. Note that this is only an optimization: correctness would be maintained regardless, as any duplicates are identified and ignored. The recovery process also takes care of non-deterministic events such as timer events and cache reads, as recorded in the interleave log, by waiting for events when necessary to preserve ordering of events and messages.

3.3.1 Local Recovery

In order to provide fast local failover, we introduce a *local recovery service*, which is responsible for storing snapshots, 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 that it can reconstruct the corresponding TCP sessions and extract CESSNA data plane messages to construct its local copy of message logs and the interleave log.[4] It also receives snapshots directly from the edge agent and stores them.

Hot Backup For applications that require very fast recovery, CESSNA provides a 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 snapshot that is taken. In case of a failure, the alternate edge is ready to immediately fetch the replay data from the local recovery service and to execute the recovery algorithm, saving the time it takes to start a new edge application instance and to restore a snapshot. As shown in Section 6, we noticed that in some runtime platforms, the vast majority of time is spent on snapshot restoration, while CESSNA' replay mechanisms are usually much faster. There is of course a tradeoff here between cost of running a hot backup and speed of recovery process.

3.3.2 Remote Recovery and Mobility

When a client fails over or migrates to a remote edge which does not share a recovery service with the failed (or previous) edge, we delegate the responsibility for the recovery to the client and the server.

Upon taking a snapshot, the edge agent stores it locally at the local recovery service, but it may also send it to the client and/or the server to allow remote recovery. The snapshot is encrypted and signed by the edge, so the hosts cannot see

[4]We assume that if TLS is used, it is terminated before the ToR switch of the edge application, as done by Google [18] and others. ToR tap port access is a requirement for enabling local recovery.

its content or tamper with it. Each application may have different preferences here, based on its specific characteristics. The following aspects should be considered:

Snapshot size: The size of a snapshot is dependent on the runtime engine (*e.g.,* VM snapshots are very big compared to library-based snapshots as we describe in Section 5) and on the edge application itself. Bigger snapshots are more expensive to transport and to store.

Client capabilities: The two aspects to consider for the client are bandwidth from the edge and available storage at the client. For example, if clients are expected to be thin, or have expensive or low bandwidth connections, we may prefer sending snapshots to the server.

Server capabilities: While bandwidth may be less restricted to the server than to the client, it is still one aspect to consider. The other is aspect is scaling. If the server handles a large number of clients, storing snapshots them may be prohibitive or require additional resources.

Upon failure or movement, in order to restart the session, the client and the server send the most up-to-date snapshot they received (if any), their outgoing message logs, and their copy of the interleave log. The newly provisioned edge is then restored to the given snapshot, and executes the recovery algorithm.

3.4 Discovery

The client should be able to find the correct edge to connect to, based on the application it is connecting to, its location, and other factors. In our design, the client can use standard techniques as used today for service discovery, such as DNS or IP anycast [13, 30].

3.5 Client / Server Design

There is a very little difference between a client and a server in our design beyond that they may have different preferences (for example, on whether to receive snapshots from the edge or not). 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 session establishment and management, and in case of an edge failure or client migration, it reestablishes the session through another edge.

The connection handler provides the following functionality: it encapsulates outgoing packets to add the CESSNA data plane header, buffers these messages, decapsulates incoming messages and stores the received interleave log.

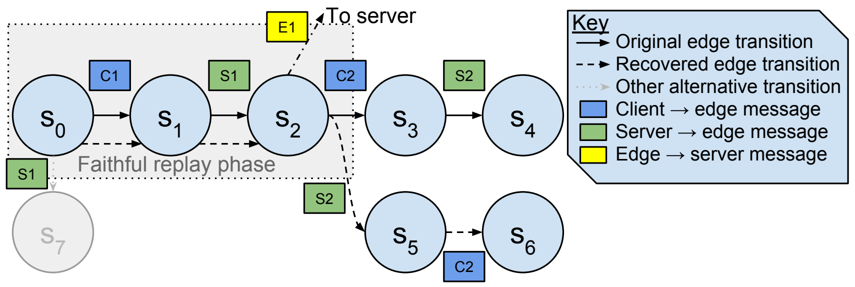

Figure 2: *Illustration of a correct replay process. Faithful replay is required only to the last committed state to guarantee output message consistency.*

4 Formal Guarantee

4.1 Formal Problem Definition

We define an edge application instance (*i.e.,* per session) as a state machine $E = \langle \mathbf{S}, s_0, M_{in}, M_{out}, \delta \rangle$, where $\mathbf{S}$ is the set of states, $s_0 \in \mathbf{S}$ is the initial state, M_{in} and M_{out} are the sets of possible incoming and outgoing messages respectively,[5] and δ is a state transition function defined as $\delta : (\mathbf{S} \times M_{in}) \rightarrow (\mathbf{S} \times M_{out} \cup \{\epsilon\})$. Note that sets may be infinite, and that δ may cause an emission of a message from M_{out}.

We denote by $\delta^*(s, \vec{m})$ the result of consecutive applications of δ on all messages in $\vec{m}$, starting from state $s \in \mathbf{S}$. This result is a sequence of messages $\vec{m_{out}}$ and a single state (formally: $\delta^* : (\mathbf{S} \times \{\vec{m} = \langle m_1, m_2, \ldots \rangle | m_1, m_2, \ldots \in M_{in}\}) \rightarrow (\mathbf{S} \times \{\vec{m}' = \langle m'_1, m'_2, \ldots \rangle | m'_1, m'_2, \ldots \in M_{out}\}))$.

We would like to prove *output message consistency*, *i.e.,* show that when a CESSNA session is moved to another edge, regardless of the reason, output messages are consistent with the ones emitted before the session was moved. CESSNA provides *input message faithful replay*, so specifically we would like to prove that this is sufficient for output message consistency.

Specifically, let E_1 be an edge. Let $\vec{C} = \langle C1, C2, \ldots \rangle$ be a sequence of messages sent from the client to the edge, and $\vec{S} = \langle S1, S2, \ldots \rangle$ be a sequence of messages sent from the server to the edge. The edge E_1 processes these messages in some order $\vec{m_1}(\vec{C}, \vec{S})$ (where $\vec{m_1}$ is a function of $\vec{C}$ and $\vec{S}$, determined by E_1). Formally, it applies $\delta^*(s_0, \vec{m_1})$, ending up in some state s', while emitting messages $\vec{m_{out}}$ before the client (and possibly the server) disconnects and attempts to continue the session with a different edge E_2, by supplying it with $\vec{C}$ and $\vec{S}$.

We define the *last committed state* of E_1 as the last state E_1 has been to that emitted a message to the outer world (*i.e.,* to the client or the server). This state could have been traversed earlier than s', which was the last state of E_1 before it

[5] M_{in} may also contain nondeterministic event information, in which case instead of replaying a message, the event is handled. We continue the formal discussion with referring to both input messages and such events as "input messages", but no part of this section precludes the replay of such events instead of an input message.

was replaced. We denote the last committed state as s^{LC}. We can then split $\vec{m_1}$ into two parts: $\vec{m_1}^{\leq LC}$ for messages that were processed until reaching s^{LC} for the last time, and $\vec{m_1}^{>LC}$ for the rest of the messages (note that $\vec{m_1} = \vec{m_1}^{\leq LC} || \vec{m_1}^{>LC}$, where $||$ is the vector concatenation operator).

Definition 1. *A* faithful replay *process at E_2 would merge messages in $\vec{C}$ and $\vec{S}$ to create $\vec{m_2} = \vec{m_2}^{\leq LC} || \vec{m_2}^{>LC}$, where $\vec{m_2}^{\leq LC}$ is identical to $\vec{m_1}^{\leq LC}$, and then apply $\delta^*_{E_2}(s_0, \vec{m_2})$, suppressing output for messages that were already received by the client and the server. Since $\vec{m_2}^{\leq LC} = \vec{m_1}^{\leq LC}$, E_2 will traverse the exact same state sequence as E_1 until s^{LC}, and therefore any message it may emit from that point is consistent with messages previously emitted by E_1.*

Figure 2 depicts an example of such process. The original edge (*e.g.,* E_1) processes messages $C1, S1$ in this order, getting to state s_2 and emitting a message $E1$ to the server. The original edge then continues processing of messages $C2, S2$, in this order, ending up at state s_4 before the session is moved to a recovered edge (*e.g.,* E_2). To ensure output message consistency, E_2 must traverse states $s_0 \rightarrow s_1 \rightarrow s_2$ in the exact same order as did E_1, so it must process messages $C1, S1$ in the same order. From s_2, the order in which messages are handled is not important, and E_2 may indeed end up traversing completely different states, but the two execution paths are both valid. Since E_1 has not emitted any message after $E1$, any further message emitted by E_2 is consistent with what was received until then by the client and the server.

If additional messages were sent by the client or the server after E_1 stopped processing, these should be appended to $\vec{m_2}^{>LC}$. The interleave order of these additional messages is not important.

4.2 Correctness Proof

We further define $(s^E, \vec{m_{out}}^E) = \delta^*_E(s_0, \vec{m_{in}})$. In other words, s^E is the state of E after starting at its initial state and processing all messages in some vector of messages $\vec{m_{in}}$, and $\vec{m_{out}}^E$ is the vector of emitted messages from the same processing.

Formally, output message consistency means that a correctly recovered edge E' is expected to emit exactly the same sequence of messages that would have been emitted by the original edge E (had it not failed) if given the same sequence of input messages m^E_{in}. That is, $\vec{m}^E_{out} == \vec{m}^{E'}_{out}$.

Theorem 1. *$\vec{m_{out}}^E == \vec{m_{out}}^{E'}$ if $\vec{m_{in}}^E == \vec{m_{in}}^{E'}$. In words: a faithful replay of the input messages is sufficient for output message consistency.*

Proof. Since the output messages are a result of applying δ^*, which is the same for E and E', we want to end up at the same state after applying δ^* on the same input vector ($s^E == s^{E'}$). Both state machines start from the same state and are deterministic. There must be a contradiction if given the same sequence of input messages the two state machines end up in different states (and hence different output sequence).

□

5 Implementation

We implemented two prototypes based on CESSNA design principles. The first is a generic container-based implementation written in Python which can be used transparently with any TCP-based application without any modification to the client or the server. It uses an off-the-shelf Docker runtime engine for edge processing and snapshotting/recovery. However, being based on a container engine not optimized for our usecase, it incurs significant performance penalties, especially for recording and restoring snapshots.

In order to reduce this overhead incurred by Docker's runtime engine, we built a specialized runtime engine that provides the same edge application API, but uses the Rust programming language. The Rust runtime engine is optimized for fast snapshotting and recovery.

We present the details of the two implementations (which we will make available with the publication of this paper) in the following subsections. In total, the code consists of 4556 lines of Python code and 5600 lines of Rust code for both implementations together, including sample edge applications (counted using CLOC).

5.1 Container Isolation-CESSNA

Container Isolation (CI)-CESSNA allows any application that uses TCP sockets to run with CESSNA. Client and server applications require no modifications to use CI-CESSNA. Edge applications are written in Python using the provided edge application API (described in Section 3.2.3). In this section, we provide details on key parts of this implementation of CESSNA.

5.1.1 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. The code contacts the host agent with the requested address. If the address 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.

5.1.2 Host Agent

Our host agent is implemented in Python. It is responsible for receiving snapshots (if desired by a host) and for managing session lifecycles. The host agent communicates with its corresponding edge agent out-of-band, in parallel to the application session.

In each host, the host agent is also responsible for spinning up a TCP proxy for each session. 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.

The TCP proxies serve as the local CESSNA connection handlers, as described in Section 3: they implement the CESSNA data plane protocol, and provide the host agents with the outgoing message logs and interleave logs extracted from incoming messages and which are necessary for future remote recovery.

5.1.3 Edge Platform and Agent

The edge platform is based on a Python edge agent that runs adjacent to the Docker engine. The edge agent manages snapshots and communication with the host agents. It may run on a different physical machine than the Docker engine, and it 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. The agent takes care of mounting filesystems for the process, and sets up port forwarding (so that multiple clients can run multiple instances of the same application in parallel). The edge agent also has an API by which an edge application can query information about the system, request that a snapshot be taken, and so on.

Edge Application API Edge applications make use of a CESSNA edge library which provides the API described in Section 3.2.3. It also implements the recovery algorithm shown in Algorithm 1, and this is triggered automatically when the edge application is started in recovery mode. The edge library also provides additional methods for managing an application's lifecycle (e.g., initialize, shutdown). We believe porting this library to other programming languages will be straightforward.

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).

Snapshots The edge agent is responsible for taking snapshots when requested by an application. Upon receiving a snapshot request, the edge agent invokes Docker's `checkpoint create` command which pauses the container, takes a snapshot, and then resumes the container. We measure the latency associated with this process in

Section 6. When this command finishes, the edge agent verifies that the snapshot was successfully taken, and responds to the edge application so that it can continue its operation normally. The snapshot files are then compressed and, depending on configuration, sent to the required remote destination(s).

5.1.4 Control Plane Protocol

The client, edge, and server agents communicate with each other using a REST API over HTTP. Messages are encoded with JSON. The protocol is simple and contains messages to request and approve a new session, to recover an existing session, to initiate a snapshot, and to close a session.

5.2 Software Isolation-CESSNA

Software Isolation (SI)-CESSNA trades application generality for improved application snapshot and recovery performance.

Edge applications that run on top of SI-CESSNA are written in Rust and use an extended version of the CESSNA Edge library. This library is a Rust implementation of the API presented in Section 3.2.3, with the addition of a set of data structures with explicit support for snapshotting.

For simplicity, we decided to make this implementation specifically for applications that use gRPC [14] to communicate between the end-hosts and the edge rather than raw TCP sockets. This shows that the CESSNA design is not bound to a specific transport technology.

5.2.1 Edge Platform

API for Improved Snapshot and Recovery We observe that container snapshots save not just application state, but also library and OS structures tied to the application. In order to reduce snapshotting and recovery overhead, we implemented a framework providing a programming model similar to what is offered by Dome [4]. This library provides a set of data structures for which we can compute snapshots. We checkpoint applications written in this framework by serializing and saving snapshots, and restore them by deserializing these saved snapshots. Edge applications then must use these data structures to store any state that persists across messages.

Application Runtime The edge platform of this implementation uses the specialized runtime engine we built in Rust. The runtime engine runs as a single process. As Rust is a safe language, it is still able to provide memory isolation among different applications [24, 19]. The runtime engine also provides the functionality of the edge agent (described in Section 3.2.2).

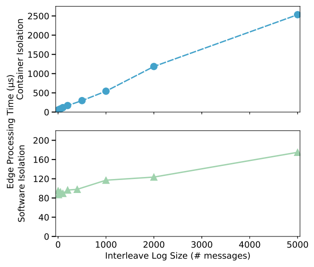

Figure 3: *Processing overhead as a function of interleave log size.*

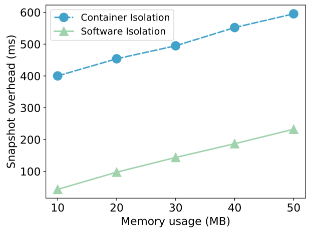

Figure 4: *Snapshotting overhead as a function of application memory usage.*

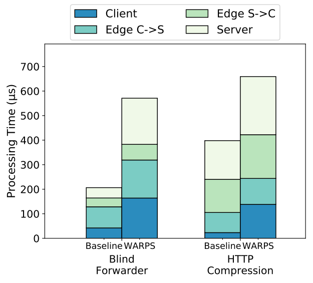

Figure 5: *Overhead of applications with CI-CESSNA.*

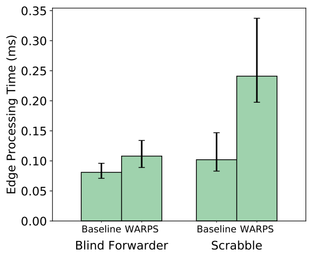

Figure 6: *Median overhead of applications with SI-CESSNA, error bars drawn at 5th and 95th percentilce latencies.*

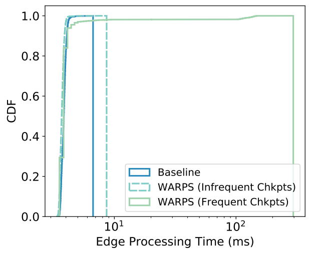

Figure 7: *CDF of SI-CESSNA edge processing times for video frame filtering for different checkpointing frequencies.*

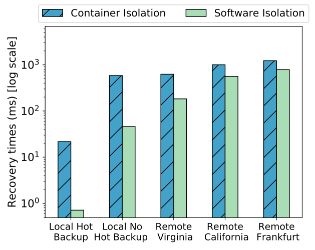

Figure 8: *Latency overhead of the recovery process when the client, edge, and server are all in Virginia and remote recovery is done to a different edge.*

Edge Application API We provide applications the same API as described in Section 3.2.3. In addition, the API provides the interface for maintaining our snapshottable data structures, which are managed automatically. The recovery process is also handled by the Edge API in case the application starts in a recovery mode.

5.2.2 Host API

We provide an API for the client and server that allows them to establish a session and send and receive gRPC messages. The host API also includes modules to provide the functionalities of the host agent as described in Section 3.5. This is different than in the generic implementation, as client and server code needs to be modified to use our API for gRPC.

6 Evaluation

We deployed CESSNA on multiple AWS nodes in different locations worldwide to measure the benefits and overheads of the system. We deployed edge nodes in Virginia, and servers in Virginia, Frankfurt, and Singapore. We used additional machines in Oregon and Virginia as clients.

6.1 Overhead of Fail-Free Operation

In this section, we evaluate the performance overheads of our two prototype implementations of CESSNA under normal operation, without failures. There are several factors that can affect the overhead of CESSNA under normal operation, without failures, and we measure the effect of each one of them.

Message Log Size Our experiments showed no significant computational overhead caused by message log growth. We measured the overhead with up to 10 k messages stored in the log. However, it should be considered that the log resides on both the client and server, and consumes memory with respect to message size, which could impose additional constraints depending on the resources available to the client or server.

Interleave Log Size The interleave log grows when the edge receives more packets than it emits. Figure 3 shows the processing overhead that is imposed by CESSNA in the edge, as a function of the size of the interleave log that is being attached to each outgoing packet. Interleave log size impacts edge performance, though only when emitting a packet after the log has grown substantially.

Snapshot Overhead Both the message log and interleave log are purged upon taking a snapshot, so the overheads above can be capped. Figure 4 shows the latency overhead of taking a snapshot in both implementations. In CI-CESSNA, at least 88 % of the time is spent on executing Docker's `checkpoint create` command. This percentage grows with the memory usage of the application. An additional 38–65 ms are due to CESSNA, for verifying the snapshot and for the communication between the edge application and the edge agent. SI-CESSNA reduces the snapshot latency overhead by 351–365 ms. In both implementations, snapshot latency overhead grows linearly with the memory size of the process.

6.2 Applications

We developed a range of sample applications with different characteristics and behaviors, atop both CI-CESSNA (Section 5.1) and SI-CESSNA (Section 5.2). SI-CESSNA applications are written in Rust. For CI-CESSNA, the edge portion is written in Python, but the client and server can be in any language, so these portions use a mixture of C, C++, Python, and Java.

For comparison, we created a native socket/gRPC-based API for each environment, which runs without CESSNA. This native API is used as the baseline for comparison. In CI-CESSNA, we run native applications inside Docker containers for adequate comparison. Applications implemented in CI-CESSNA are denoted by

* and applications implemented in SI-CESSNA denoted by †. The applications we developed are as follows:

Blind Forwarder *† A simple edge application that forwards every message it receives to the other side of the edge (e.g., from client to server and vice versa).

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 [21]), and tested it with an unmodified Apache Tomcat 7 web server and an unmodified web-browser.

Figure 5 shows the latency of each hop (client, edge, and server), for the above two applications, with and without CI-CESSNA, for a single roundtrip client-edge-server-edge-client. CESSNA cumulative overhead for a roundtrip is below 364 µs (comparing medians). This should be compared to the 5–20 ms latency expected from next generation edge frameworks [2] and with link latencies in the order of milliseconds and higher (*e.g.,* we measured 9–30 ms latency between AWS sites in continental US (depending on their distance) and 43–118 ms between the US and Frankfurt/Singapore).

Multi-Player Games *† We implemented Battleship* and Scrabble† 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.,* that the user chose valid words in Scrabble, or that they hit or miss a ship in Battleship), and synchronizes the game state with the server. Figure 6 shows the median edge processing latency for Scrabble implemented on SI-CESSNA. SI-CESSNA adds 150 µs of edge processing time, due to the persisting of board state updates to disk after receiving every message. The client and server processing times are omitted; the overhead imposed by SI-CESSNA on these components is minimal.

In addition, we created edge applications for two existing open-source multiplayer games – Pong† and Snake*. The edge applications we created hide game dynamics from the clients. They receive keystrokes from the client and send back rendering information (object positions, etc.). All rule checking and game object rendering is done at the edge. This improves latency and reduces computation at the client. The difference between these two games and the previous ones is that they are much more dynamic and similar to real-life multiplayer games: players move all the time, simultaneously, and hundreds of messages are sent between the client, the edge, and the server each second.

Video Frame Filtering † This edge application filters out redundant video frames by computing a perceptual hash [31] over the stream of frames sent by the client, saving bandwidth from the edge to the server. Our implementation filtered out 51 % of the frames in a video stream captured by a traffic camera. Figure 7 shows the variation in performance that checkpointing frequency introduces. If an application frequently requests checkpoints, the tail latencies increase due to computing state change accounting and persisting checkpoints to disk.

6.3 Recovery

Figure 8 shows the latency overhead of the recovery process, for both local and remote recovery.

Container Isolation-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 without a hot backup incurs 585 ms overhead. The additional overhead is mainly due to Docker's checkpoint restore command (68 %), while CI-CESSNA agent incurs another 27 % for preparing the snapshot, decompressing it, and verifying the recovered container before resuming the session. Remote recovery adds link latencies as expected.

Software Isolation-CESSNA There is a substantial improvement in recovery: the latency overhead for local recovery with a hot backup is reduced by 29 × to 0.71 ms (median result), while the overhead for local recovery without a hot backup is reduced by 12 × to 45.95 ms when restoring a checkpoint of a similar size as used for CI-CESSNA. Remote recovery in the same AWS region has 182.9 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 14 µs per replayed message.

7 Discussion

7.1 Applicability

Many applications that benefit from the client-edge-server model do not have a stateful edge, or do not have strong correctness requirements on their edge state. For other applications, however, maintaining state correctness during cases of failure or migration may be vital. One class of such applications, as was discussed in Section 1, is online multiplayer games. For instance, we showed in Section 6 four edge-based online games where the edge holds the current state of the game, from the client's perspective. This state is updated based on clients' moves with low latency response to the client, and also based on other players' moves. These latter ones are coordinated and sent to the edge by the server. If such an application does not maintain this state perfectly under failover or client mobility, players would have state that is inconsistent with their prior actions. This will, at the very least, put players in a confusing situation. In the worst case, it will create fatal errors.

It is applications in this class – stateful applications which benefit from correctness during failure or migration – for which CESSNA is ideally suited. This raises two points. First, CESSNA does not *force* an application to use stronger guarantees than it needs. Many real-world client-edge-server applications may contain several components, some that require correctness guarantees and some that do not. One can choose to use CESSNA for only the portion of the application that falls into the former class. For example, while a video conferencing application does not require any correctness guarantees for video frames, it may require some for the control channel that keeps the call active, so calls would not drop in case of an edge failure (and rather just hang momentarily). Second, it is certainly possible to write applications with seamless failover without CESSNA. However, doing this on a per-application basis (and, in particular, getting it *right*) is typically nontrivial. CESSNA factors out this aspect of the design and provides a general solution.

7.2 Handling Multiple Clients per Session

Our current design does not allow an edge application to handle more than a single client per session. The multiple clients case is not only harder to solve, it is even harder to define. For example, in such a case, one may ask what consistency guarantees should be given when a client joins in the middle of a session, or when one of several client leaves (or fails) during a session.

Additionally, we claim that if edge state can be shared by multiple clients, at least some of them must lose the latency benefit of using an edge, and therefore might prefer to connect directly to a server. We formally define this problem and prove it in Appendix A.

However, we note that our basic mechanisms are sufficient for correct recovery from edge failures even in the case of multiple clients if we assume that all clients (and the server) start and finish the session at the same time with none leaving or failing during it. Under these conditions, the CESSNA design could be applied with a few modifications.

We further note that even in the normal case where CESSNA cannot be used to share state between clients, multi-client applications that share state at the server can still benefit from CESSNA, for example by having a closer edge that provides stateful compression or rendering (as we show in the gaming examples in Section 6).

8 Related Work

FTMB [28] presented a framework for rollback recovery for middleboxes. The problem of middlebox recovery is quite different than edge recovery for several reasons: nature of state, sources of traffic, and transparency (clients and servers may not know about the middleboxes, and may not cooperate with them). The main difference in our case lies in the fact that an edge in CESSNA processes one message at a time, and we expect applications to use explicit locks for concurrent mutable accesses to data structures. This greatly simplifies and speeds up the replay mechanism used in CESSNA. Furthermore, we target a different workload since we focus on application-level functionality rather than on packet processing. While adopting FTMB's mechanisms directly might allow CESSNA to support a larger variety of edge semantics, it would drastically increase the size of the replay log (which must now include ordering information for each access), and thus render our choice to send this log to the client and server impractical. As a result, both designs are complimentary and apply to different settings.

Remus [7] and Colo [10] are two no-replay solutions that provide fault tolerance for any VM-based system, either using checkpointing and output buffering, or by running redundant VMs simultaneously. Other log-based rollback recovery protocols have seen only little adoption in real systems due to their complexity [12]. Several other works addressed virtual machine recovery, either for single core processors [5] or for multiprocessors [11]. State replication requires multiple hot standby instances to provide survivability, and the state within these must be updated before any messages affected by the state are emitted. This additional coordination imposes a performance penalty during failure-free operations and greatly limits where such active standbys can be placed.

The OpenFog Reference Architecture [23] describes an architecture for fog computing, which can be seen as a superset of the client-edge-server model on which we focus. The architecture covers many deployment, management, and operational concerns which are likely relevant to any fog/edge system, but are orthogonal to CESSNA' focus on providing correctness guarantees during edge failover. Similarly, previous work [17] has evaluated the use of Docker as an enabling technology for edge computing; while the choice to use Docker in our Container Isolation CESSNA prototype is motivated by some of the same observations and conclusions as in that work, our contribution is orthogonal. Another line of work [26, 25] has evaluated using VMs for creating minimalist cloud infrastructure at the edge, but they are intentionally limited to stateless applications due to the lack of recovery mechanisms. CESSNA can be used to enable stateful applications on such frameworks.

EdgeComputing [8] was an early attempt by Akamai to provide stateful edge computing, specifically for web applications. It uses state replication to provide fault tolerance and to support mobility. However, despite being a commercial product, customers failed to adopt it due to its complexity and limitations [22]. Currently, Akamai mostly markets Cloudlets which are stateless, as EdgeComputing adoption has been low due to customers finding them hard to program [29].

A recent workshop paper [16] was the first to define the consistency problem solved in this paper. That paper only presented the problem and a preliminary and limited design. It does not allow applications to use non-deterministic operations or multithreading in the edge at all. That paper also did not have an implementation or evaluation, and did not provide formal guarantees. Our paper provides all of these, a much improved design, and two reference implementations.

9 Conclusion

This paper proposes a framework that provides strong correctness guarantees for stateful network edge applications. Such applications allow offloading of computation from clients and servers, reduction of response latency, and reduction of backbone utilization. While edge computation is already a fact, its correctness under failure and mobility is not guaranteed by current approaches.

Our proposed model is general enough for many types of applications, and yet is feasible for actual implementation and deployment. 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, the performance overheads can be significantly lowered.

Acknowledgements

This work was done in collaboration with Yotam Harchol, Vivian Fang, James Murphy McCauley, Marc Korner, Aurojit Panda, and Scott Shenker.

I thank my advisors, Scott and Sylvia, for being supportive and helping me reach my potential. I thank Yotam for his guidance and mentorship that has helped me become a better researcher. I thank my husband, Asad, who has been a great support and most likely has an equal contribution to this work as me.

References

[1] Amazon Web Services. AWS Lambda Programming Model, 2018. `https://docs.aws.amazon.com/lambda/latest/dg/programming-model-v2.html`.

[2] Arutperunjothi, R. Akraino edge stack, 2018. `https://wiki.akraino.org/display/AK/Akraino+Edge+Stack`.

[3] Barroso, L. A., Dean, J., and Hölzle, U. Web search for a planet: The google cluster architecture. *IEEE Micro 23* (2003), 22–28.

[4] Beguelin, A., Seligman, E., and Stephan, P. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing 43*, 2 (1997).

[5] Bressoud, T. C., and Schneider, F. B. Hypervisor-based fault-tolerance. In *SOSP* (1995).

[6] Corbet, J. Checkpoint/restart in userspace. LWN `https://lwn.net/Articles/572125/`, 2013.

[7] Cully, B., Lefebvre, G., Meyer, D. T., Feeley, M., Hutchinson, N. C., and Warfield, A. Remus: High availability via asynchronous virtual machine replication. In *NSDI* (2008).

[8] Davis, A., Parikh, J., and Weihl, W. E. EdgeComputing: Extending enterprise applications to the edge of the internet. In *WWW Alt.* (2004), pp. 180–187.

[9] Docker checkpoint and restore. `https://github.com/docker/cli/blob/master/experimental/checkpoint-restore.md`, 2018.

[10] Dong, Y., Ye, W., Jiang, Y., Pratt, I., Ma, S., Li, J., and Guan, H. COLO: coarse-grained lock-stepping virtual machines for non-stop service. In *SOCC* (2013).

[11] Dunlap, G. W., Lucchetti, D. G., Fetterman, M. A., and Chen, P. M. Execution replay of multiprocessor virtual machines. In *VEE* (2008).

[12] Elnozahy, E. N., Alvisi, L., Wang, Y., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*, 3 (2002).

[13] Freedman, M. J., Lakshminarayanan, K., and Mazières, D. Oasis: Anycast for any service. In *NSDI* (2006).

[14] GRPC: A high performance, open-source, universal RPC framework. `https://grpc.io`, retrieved 01/21/2017.

[15] Guerraoui, R., and Schiper, A. Software-based replication for fault tolerance. *IEEE Computer 30* (1997), 68–74.

[16] Harchol, Y., Mushtaq, A., McCauley, J. M., Panda, A., and Shenker, S. CESSNA: resilient edge-computing. In *MECOMM@SIGCOMM* (2018), pp. 1–6.

[17] Ismail, B. I., Goortani, E. M., Karim, M. B. A., Tat, W. M., Setapa, S., Luke, J. Y., and Hoe, O. H. Evaluation of docker as edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)* (Aug 2015).

[18] Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., Bailey, J., Dorfman, J., Roskind, J., Kulik, J., Westin, P., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., Chang, W.-T., and Shi, Z. The QUIC transport protocol: Design and internet-scale deployment. In *SIGCOMM* (2017), pp. 183–196.

[19] Levy, A., Campbell, B., Ghena, B., Giffin, D. B., Pannuto, P., Dutta, P., and Levis, P. Multiprogramming a 64kb computer safely and efficiently. In *SOSP* (2017), pp. 234–251.

[20] Maynard-Koran, P. Fixing the internet for real time applications: Part ii, 2016. `https://engineering.riotgames.com/news/fixing-internet-real-time-applications-part-ii`.

[21] NGINX Inc. Compression and decompression, 2019. `https://docs.nginx.com/nginx/admin-guide/web-server/compression/`.

[22] Nygren, E., Sitaraman, R. K., and Sun, J. The akamai network: A platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev. 44*, 3 (Aug. 2010), 2–19.

[23] OpenFog Consortium Architecture Working Group. OpenFog Reference Architecture for Fog Computing, Feb. 2017. `https://www.openfogconsortium.org/ra/`.

[24] Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S., and Shenker, S. NetBricks: Taking the V out of NFV. In *OSDI* (2016), pp. 203–216.

[25] Satyanarayanan, M. The emergence of edge computing. *IEEE Computer 50*, 1 (2017), 30–39.

[26] Satyanarayanan, M., Bahl, P., Cáceres, R., and Davies, N. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing 8*, 4 (2009), 14–23.

[27] Schneider, F. B. Replication management using the state machine approach.

[28] Sherry, J., Gao, P. X., Basu, S., Panda, A., Krishnamurthy, A., Maciocco, C., Manesh, M., Martins, J., Ratnasamy, S., Rizzo, L., and Shenker, S. 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).

[29] Sitaraman, R. Personal communication, 2018.

[30] Wang, L., Pai, V. S., and Peterson, L. L. The effectiveness of request redirection on cdn robustness. In *OSDI* (2002).

[31] Yang, B., Gu, F., and Niu, X. Block mean value based image perceptual hashing. In *Intelligent Information Hiding and Multimedia Signal Processing, 2006. IIH-MSP'06. International Conference on* (2006), IEEE, pp. 167–172.

[32] Zhang, I., Denniston, T., Baskakov, Y., and Garthwaite, A. Optimizing vm checkpointing for restore performance in vmware esxi. In *USENIX Annual Technical Conference* (2013).

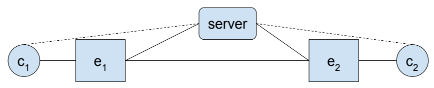

Figure 9: *Example of network graph.*

A On the Impossibility of Edge State Sharing by Multiple Clients

In this section we elaborate on our claim that when state can be shared by multiple clients at the edge, at least some of the clients lose the latency benefits of having an edge (and hence might prefer not to connect through the edge, but directly to the server). In the rest of this section we define the problem formally and prove our claim.

We represent the logical network of edges and clients as an undirected graph where clients $\{c_1, c_2, \ldots\}$, edge nodes $\{e_1, e_2, \ldots\}$, and the server s are vertices and logical links between them are graph edges. Each logical link (graph edge) between nodes u and v has a latency value $\ell_{(u,v)} > 0$. An example of such graph is shown in Figure 9. For simplicity, we assume that for any two nodes u, v, $\ell_{(u,v)} = \ell_{(u,v)}$, but the assumptions and following proof can be easily extended to support directed graphs in which this property does not hold.

Before formulating the main theorem, we present the following assumptions:

1. There is at least one state object that is shared for more than a single client, and it is stored on one or more edge nodes.
2. When a client c reads or modifies the state object by communicating with an edge e, it expects the latency to be lower or at most equal to $\ell_{(c,s)}$. Otherwise it has no latency benefit in using that edge.
3. For any given edge node e_1, there exist at least one other edge node e_2 for which the latency $\ell_{(e_1,e_2)}$ is greater than $\ell(e_1, s) + \ell_{(e_2,s)}$.
4. Not all clients that share the state connect to the same single edge. We assume that because, first, it is realistic, and second, even in a non-realistic extreme case, clients may still move between edges (as we support client mobility), so this situation must be taken into account in any case.
5. Triangle inequality: for each client c that is connected through edge e, $\ell_{(c,s)} \leq \ell_{(c,e)} + \ell(e, s)$ (even if the edge is connected through faster links, the edge can be used just for forwarding).

Theorem 2. *In any configuration, for at least one client, it is impossible to satisfy all these assumptions. Specifically, assumption 2 does not hold.*

Proof. We prove the theorem by contradiction: we assume that assumption 2 holds for all clients.

We focus on two clients that connect through different edges and share some state object. Two such clients must exist due to assumption (4). We denote them c_1, c_2, and we denote the edges they connect to as e_1, e_2, respectively. Based on assumption (3), we assume e_1 and e_2 are those two edge nodes for which $\ell_{(e_1,e_2)} > \ell(e_1, s) + \ell_{(e_2,s)}$.

The shared state may be stored solely on e_1, solely on e_2, or on both of them (*i.e.,* using some replication technique). We consider a scenario where a write operation that originates from one client's request, and modifies the state, is followed by a read operation, originated from a request of another client.

Without loss of generality, let c_1 be the one to initiate the write, and c_2 be the one to initiate the read.

If the state is stored solely on e_1, then the latency for c_1 is $\ell_{(c_1,e_1)}$ (assuming no additional latency is incurred). Next, c_2 issues a request to its edge, e_2, that requires reading the shared state from e_1. Thus, the latency for c_2 is at least $\ell_{(c_2,e_2)} + \ell_{(e_2,s)} + \ell_{(e_1,s)}$, and this is greater than $\ell_{(c_2,s)}$ (also based on assumption (5)). However, this contradicts assumption (2).

The same goes for the case when the state is stored solely on e_2, by switching the roles between c_1 and c_2.

If the state is replicated on both e_1 and e_2, then the replication must provide strict consistency: any write to one edge must either invalidate the replication on the other edge (thus require the next read operation to wait until a fresh copy is fetched – reader's latency is $\ell_{(c_{reader},e_{reader})} + \ell_{(e_{reader},s)} + \ell_{(e_{writer},s)} > \ell_{(c_{reader},s)}$), or proactively update the replication (writer's latency is $\ell_{(c_{writer},e_{writer})} + \ell_{(e_{writer},s)} + \ell_{(e_{reader},s)} > \ell_{(c_{writer},s)}$), both contradicting assumption (2). □