CESSNA: Resilient Edge-Computing

Yotam Harchol
UC Berkeley
yotamhc@berkeley.edu

Aisha Mushtaq
UC Berkeley
aisha@berkeley.edu

James McCauley
UC Berkeley
jmccauley@cs.berkeley.edu

Aurojit Panda
NYU and ICSI
apanda@cs.nyu.edu

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

ABSTRACT

The introduction of computational resources at the network edge has moved us from a Client-Server model to a Client-Edge-Server model. By offloading computation from clients and/or servers, this approach can reduce response latency, backbone bandwidth, and computational requirements on clients. While this is an attractive paradigm for many applications, particularly 5G mobile networks and IoT devices, it raises the question of how one can design such a client-edge-server system to tolerate edge failures and client mobility. The key challenge is to ensure correctness when the edge processing is stateful (so the processing depends on state it has previously seen from the client and/or server). In this paper we propose an initial design for meeting this challenge called Client-Edge-Server for Stateful Network Applications (CESSNA).

CCS CONCEPTS

• **Networks** → *Programming interfaces*; *Network reliability*; *Network mobility*;

ACM Reference Format:
Yotam Harchol, Aisha Mushtaq, James McCauley, Aurojit Panda, and Scott Shenker. 2018. CESSNA: Resilient Edge-Computing. In *MECOMM'18: ACM SIGCOMM 2018 Workshop on Mobile Edge Communications , August 20, 2018, Budapest, Hungary*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3229556.3229558

1 INTRODUCTION

The recent introduction of compute and storage resources at the network edge allows service providers to offer lower latency and higher throughput to geographically nearby content and computation, and this in turn allows applications such as sensors and IoT devices to reduce their upstream bandwidth requirements by pre-processing data at the edge.

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 consistency issues of the client-server model have been thoroughly studied, the addition of a stateful edge processor in between the two complicates the consistency problem.

To illustrate the problem, consider a simple example of an edge that serves as a packet counter: The server is not interested in getting every packet from the client, but only in the total number of packets the client has emitted. The edge is thus holding the counter value and if the edge fails, even if another edge is brought up immediately, the state is lost and the server never gets an accurate count of the emitted messages.

This is merely an illustration of a general problem. In this paper we first formally articulate the edge consistency problem, and then propose a general purpose framework for client-edge-server applications that provides strong consistency guarantees as described later in this paper.

Our computational model, elaborated in Section 2, is of a computationally-capable edge that allows offloading of computation from the server, the client, or both (and can receive messages from both clients and servers). We assume that the edge is stateful but keeps state on a per-client basis: that is, a new edge process (or set of processes) is instantiated to handle each client-server session. Thus, the consistency we wish to provide for the edge would guarantee that the edge's state always correctly reflects both client's and server's inputs (relative to this session) for as long as this session is active. Moreover, we only care about preserving consistency in the case of an edge failure; if the client or server fails, we assume the session terminates implicitly. However, note that nothing in our design precludes the use of replication or other techniques to increase the resiliency of the server (or even the client).

Given this model, we would want a framework that will provide client-edge-server applications with these consistency guarantees, even though the edge may arbitrarily fail, and clients may arbitrarily move between edges. Our design aims at no or minimal modification to the source code of existing applications.

Our proposed design is built in two layers, for two different types of edge recovery: *local recovery*, and *remote recovery*. Local recovery can be used when the failed edge and the recovered edge are physically close to each other, for example, under the same ToR switch. In this case we use a mechanism similar to the one used in [12], though we strip it down to a much-simplified approach, which is enough due to the client-specific nature of edge state that we consider.

MECOMM'18, August 20, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5906-1/18/08…$15.00
https://doi.org/10.1145/3229556.3229558

Remote recovery refers to the case when the two edges are far from each other, and can also apply to the client mobility case. In this case, the client and the server cooperate with the newly provisioned edge to quickly restore its state and continue the session from where it has stopped.

We make the following observations when coming to design such a framework:

- The edge receives messages from two different parties, and its state may be dependent on the exact ordering of these messages. Thus, for any process of reconstruction of the state, we must have this ordering available.
- In order to allow such a reconstruction of the edge's state, each endpoint (client, server) should keep a copy of the messages it sent to the edge, at least until the edge can guarantee these copies are not needed anymore.
- Since the edge is not reliable, the ordering of incoming messages must be stored elsewhere. One option is to send it to either the client or the server. Since each outgoing message from the edge, to either the client or the server, may indicate some state change at the edge, each such message should be accompanied with the incremental addition to the edge's total incoming message ordering. Another option is to store it remotely. We intermix the two options in our design as we describe later.

Of course, actually storing all outgoing messages forever may be prohibitive for most applications in terms of memory, and may have significant and negative performance implications in cases of failure. Thus, we use periodic snapshots in order to limit the size of the required buffers, and reduce the time for session reconstruction. We describe the design in detail in Section 3.

2 COMPUTATIONAL MODEL

Having computation at the edge allows one to (i) offload computation from the client (so it can be weak and/or low-powered), and/or (ii) offload computation from the server (so that responses to the client can have lower latency), and/or (iii) reduce bandwidth to the server (by doing preprocessing at the edge). Thus, the edge can be seen as extending the power of the client and/or extending the reach of the server. As a result, one cannot think of the edge as merely splitting the client code, or merely splitting the server code, but could involve a bit of both.

We assume the purpose of a system is to process inputs coming from clients and the server. This processing can result in packets being emitted to the server, or to the client (or both). Thus, the *logical model* is one of clients sending input to the system, and perhaps receiving responses, or updates from the edge, presumably based on input from the server.

Servers. We assume that clients communicate with the system by logically sending messages to a server. This is done via the edge. The backend system handles all issues of replication and recovery for these servers and any other backend processing. There are many options here, and we leave this up to the application designer. Similar to the current client-server model, a single server can service several (or all) clients, can store data in a database, and allow clients to coordinate among each other. We place no restrictions on how clients communicate and coordinate through a server, and only require that the server be able to play back messages.

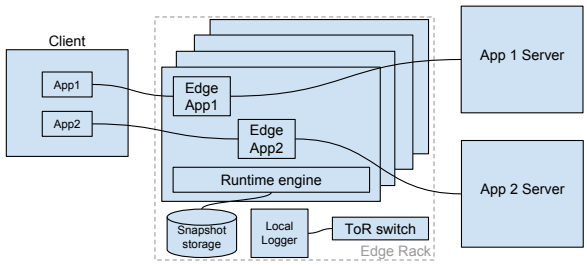

Figure 1: The general design of our framework.

Clients. We assume that clients do not depend on detailed timing information between messages or on latency of message response. Beyond this we allow clients to perform arbitrary processing, and to depend on arbitrary input including input from external sensors, video cameras, game controllers, etc. Finally, we assume that clients can be mobile, and as a result they might connect to different edges over time. Thus, applications may not assume that a particular set of clients is connected to a common edge. For applications where such aggregation is desirable and where clients are immobile (such as applications which aggregate inputs from multiple sensors [1]), we treat the set of clients whose input is being aggregated as a *single* logical client.

The Edge. Clients send a series of messages to the edge, which in turn can send messages to the client and messages to the backend server. The edge also receives messages from the backend server, and can use those in its processing of messages (for example, these messages may cause state changes in the edge). We assume that the edge application correctly and consistently handles inputs during such state changes in the absence of failures (correct behavior in the presence of failures will then be provided by CESSNA automatically). In particular, we require that the edge application be designed so that state updates are atomic and a single message (or packet) is processed using only one version of the state.

2.1 Problem Statement

Given the above assumptions, we would like to design a consistency framework for a stateful edge, such that in case of a failure of an edge instance, another instance can be provisioned and the state is correctly recovered.

The correctness of the recovery process is defined such that the recovered edge continues to process input messages and emit output messages exactly the same as the original edge would have. If there was more than one plausible outcome for the original edge at the time of the failure, the outcome of the recovered edge must be one of these plausible outcomes.

3 FRAMEWORK DESIGN

The design of our framework is illustrated in Figure 1. In this section we discuss the design of each component in the figure.

3.1 Edge Design

The main contribution of this work is a design for an edge runtime environment that allows seamless local and remote failover of edge applications, while preserving the correctness of the state at the

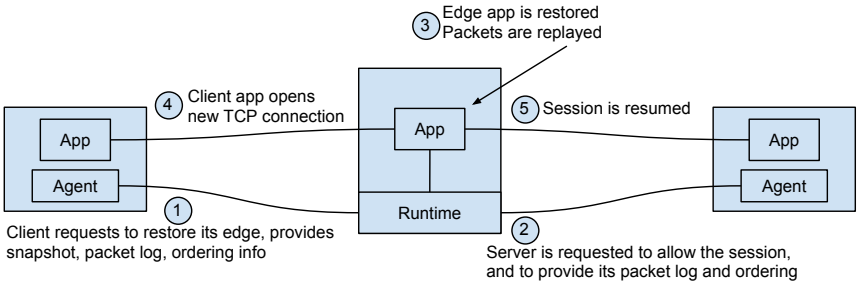

Figure 2: Illustration of the remote edge recovery process. Circled numbers indicate the order of events.

edge, such that the entire failover process is transparent to the client and the server applications.

3.1.1 Runtime Engine. Edge applications are software, or more precisely, processes. They can run in virtual machines or containers, with some hypervisor underneath. Our design does not require any specific runtime engine or hypervisor. We only require it to provide the following features:

- **Generic software:** We would like the edge to be able to run generic software as much as possible, without limiting it to specific programming languages or uncommon libraries.
- **Efficiency:** The runtime engine should allow efficient running of multiple applications on top of it, in parallel.
- **Snapshotting:** The runtime engine should be able to take snapshots of running instances and to restore an instance given such a snapshot.

Examples for existing products that provide these features are Docker [6], KVM [3], VMware [13], etc.

3.1.2 Edge Storage. Each edge runtime has some shared storage capabilities. This storage may be used by multiple instances of the same edge application when multiple client sessions benefit from sharing data – a prime example of this being content caching. The storage can be shared with instances on the same physical server, same rack, etc.

Note that the shared storage must not be used for state-related storage. The state of each instance must be managed in memory for each instance, as snapshots do not include data from the shared storage. The state of an edge application must not be dependent on the presence, or the lack of presence, of a specific item in the shared storage.

3.2 Edge Recovery

Our recovery model assumes that the edge application is fully stateful and that its state is a function of both the client and the server. If the edge application is stateless, or if its state is only a function of one side of the communication (client or server), then the recovery model can be much simplified. We discuss these simplifications after presenting the fully stateful model.

The recovery model has two layers: local recovery, which refers to the case when the replacement edge has relatively fast storage shared with the failed edge, and remote recovery, which refers to the case when the two edges are distant from each other. We make this distinction since the local case can be solved using existing techniques as we discuss below, while for the remote case a more complex model is required.

3.2.1 Local Recovery. In order to provide fast local failover, we use a simplified version of the technique presented in FTMB [12]. In FTMB, the framework is able to recover arbitrary network functions by logging and sequencing their incoming and outgoing packets, and their nondeterministic decisions, together with periodic snapshots.

In our case the situation is simpler: we treat sessions as logically independent entities, so we do not need to account for sequencing information across flows or sessions. Furthermore, we assume that the edge application is deterministic and produces consistent output given identical inputs, and thus do not log outgoing packets.

Because of the reasons above we also do not need to pause the application while taking a snapshot. If the underlying runtime supports live snapshotting, we can simply store the last position in the incoming packets log, then take a snapshot, and store the two together. Upon recovery, our framework would restore the snapshot and replay packets from the stored position. Since we assume traffic is sent over TCP, the application will ignore packets it has seen since the time the log position was logged until the snapshot was actually taken, as such packets simply appear to be delayed duplicates.

In order to store the snapshots and the log, the system should have some local storage. This can be per physical machine, or per rack of multiple machines, for example. This snapshot storage is not to be logically confused with the shared storage discussed in Section 3.1.2, which is used for application data storage and sharing. Physically, they can be colocated.

3.2.2 Remote Recovery and Mobility. In the case when the client fails over to a remote edge, which does not share a packet logger and snapshot storage with the failed (or previous) edge, we delegate the responsibility for the recovery to the client and/or the server.

Upon taking a snapshot, the edge runtime stores it locally, but it also sends it to one or two of the endhosts (the client and/or the server). It is only necessary to send it to one of them for correct remote recovery, and we assume that for most applications it would make sense to only send it to the client (to allow scaling at the server). The snapshot is encrypted and signed by the edge, so the client cannot see its content or tamper with it.

In addition to the snapshot, the edge also sends the endhosts information that will later help a recovered edge to determine the

order in which messages from both sides were processed by the failed edge. This is done such that every packet emitted by the edge, to either the client or the server, contains the most up-to-date information on this ordering.

Upon recovery, 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 knowledge of the ordering discussed above. The newly provisioned edge is then restored to the given snapshot, and then it orders the messages given by the endhosts based on the ordering they provided. This process is illustrated in Figure 2.

3.2.3 Stateless or Semi-Stateful Edge. The recovery mechanisms described so far assume a fully stateful edge. However these mechanisms can be simplified in the following cases:

If the edge application is completely stateless, we only need to be able to replay messages from the client and the server, which were already sent, but have not yet been processed. Thus, we only need the replay mechanisms described above (either using messages logged at the client, for a remote recovery, or using the local logger, in a local recovery).

In the case of a semi-stateful edge application, where its state is only a function of one of the endhosts (client or server), we also need to have state snapshots, in addition to the replay mechanism that is required for the stateless case. The replay from the endhost on which state is dependent should be from the first message after the last snapshot was taken. We do not need the interleaving ordering of client and server messages.

Based on the nature of the application, it can declare whether it is stateless, semi-stateful, or stateful, and the framework can then adjust its recovery mechanisms for this application accordingly.

3.3 Discovery

The client should be able to find the correct edge to connect to, based on the application it is connecting to, its location, etc. In our design, there is a discovery service that provides this information. Also, once the client is connected to an edge (e.g., the default one), this edge can provide it with alternative edge addresses, so in case of a failure, the client does not have to use the discovery service again but instead can immediately contact an alternative edge.

3.4 The CESSNA Protocol

Each message sent from each of the entities in our design should be sequenced, so that we could later refer to it in the ordering described above for remote recovery (local recovery uses TCP sequencing). Packets going out of the edge also contain ordering information to be stored at the endhosts.

In order to facilitate that, we design a simple layer-7 protocol, and we wrap all packets with its header. This header may contain just a sequence number (for messages going out of the hosts), or a sequence number and ordering information (for messages going out of the edge). This header precedes any layer-7 payload in packets.

In the current version of the CESSNA protocol, the header for messages from endhosts to the edge is 16 bytes long. The header for messages from the edge is at least 20 bytes long, depending on the frequency of messages emitted by the edge. Each message from the edge contains the differential logging of packets received by the edge. Thus, the more messages emitted by the edge, the shorter the header is. We also note that the header used by our prototype is optimized for simplicity and not size; its size could be reduced by encoding the current information using variable-length integers or by leveraging application-specific properties.

3.5 Client / Server Design

There is no actual difference between a client and a server in our design, except for their possible different set of preferences. For example, of whether to receive snapshots from the edge or not. An endhost, whether a client or a server, is simply an application running on top of our host platform, which manages the communication with the edge.

4 INITIAL IMPLEMENTATION

We have begun to implement the design described in this paper. In our implementation, we use Docker as the runtime engine for the edge. We briefly examine our prototype in the following subsections.

4.1 CESSNA Library

We design a shared library to be used by applications both at the endhosts and at the edge, to take care of the serialization and deserialization of messages using the CESSNA protocol, logging messages at the endhosts, and so on.

The runtime library is implemented in C++, and it overrides the Linux system calls for socket handling, such as `connect`, `accept`, `send`, `recv`, `close` (and several others). The library is loaded dynamically using the `LD_PRELOAD` environment variable so that applications need no modification in order to use it. This, for example, also enables Java and Python applications to use the library with no modification (as JVM and the Python interpreter use the OS socket library underneath).

4.2 Host Agent

Our host agent is implemented in Python. It is responsible for all slow path tasks at the hosts: logging outgoing packets, tracking the order of packets reported by the edge, receiving snapshots, and restarting sessions in case of a failure. The agent communicates with its corresponding edge runtimes out-of-band, in parallel to the application sessions.

4.3 Edge Platform

The edge platform is based on a Python agent that runs adjacent to the Docker engine to manage snapshots and communication with the host agents.

4.4 CESSNA Over HTTP

In addition to the described implementation, which requires the usage of the CESSNA library and the host agent on the client, we are also working on a version of CESSNA specifically suited for web applications (running over HTTP/HTTPS) which does not require the installation of an additional CESSNA agent or usage of the CESSNA library at the client.

CESSNA over HTTP is an extension to the CESSNA edge and server platforms that implements the client features in Javascript,

so that the client can participate in the backup and recovery process just as in the original design, with the web browser handling the entire logic by simply running Javascript code given by the edge or the server. This code is responsible for logging outgoing requests, storing ordering information received from the edge, and managing snapshots. It is also responsible for recovery of sessions.

5 APPLICABILITY & DISCUSSION

Many applications that benefit from the Client-Edge-Server model do not have a stateful edge, or do not have strong consistency requirements on their edge state. For example, an audio/video conferencing application may benefit from an edge which can reflect streams to other clients on the same edge, and combine and transcode data being sent "upstream" to remote clients. If an edge is transcoding a video frame for a client and the client migrates to another edge, it may be acceptable to simply drop that frame (indeed, if the time taken by the infrastructure to provision a new edge and/or the time taken by the client to establish a session with the new edge is greater than the duration of a frame, there is no point in doing otherwise).

For other applications, however, maintaining state consistency during cases of failure or migration may be vital. For instance, one might imagine an edge-based system which uses deep packet inspection to provide network-based security. DPI systems typically must track the state of various protocols; if such an application does not maintain this state perfectly under failover, connections may be aborted or policy violations may occur. It is applications in this class – stateful applications which benefit from consistency 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 consistency 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.

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. The benefit of CESSNA is that it factors out this aspect of the design and provides a general solution that applications can just use.

6 RELATED WORK

FTMB [12] 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). Nonetheless, we do adopt some of the ideas of FTMB for our local recovery design, as described in Section 3.

Remus [5] and Colo [7] 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 [9]. Our framework is tailored to a specific use case where this complexity can be significantly lowered. Several other works addressed virtual machine recovery, either for single core processors [2] or for multiprocessors [8]. Such techniques could potentially be used for our local recovery, but in practice they have significant overheads due to their generality.

The OpenFog Reference Architecture [11] describes an architecture for fog computing, which can be seen as a superset of the client-edge-server model on which we focus (e.g., fog nodes may communicate with other fog nodes rather than only with their clients and servers). The architecture covers many deployment, management, and operational concerns which are likely relevant to any fog/edge system, but are orthogonal to CESSNA's focus on providing a computational framework that allows for consistency guarantees during edge failover. Similarly, previous work [10] has evaluated the use of Docker as an enabling technology for edge computing; while the choice to use Docker in our CESSNA prototype is motivated by some of the same observations and conclusions as in that work, our contribution is orthogonal.

FADES [4] is a virtualization framework for offloading single-purpose tasks to the edge, specifically tailored for IoT. This is again orthogonal to our contribution, as FADES does not provide any correctness guarantees, and CESSNA may be used within such an environment for this purpose.

7 CONCLUSION

In this paper we propose an initial design for a framework that provides consistency guarantees for stateful network edge applications. Such applications allow offloading of computation from clients and servers, reducing response latency, backbone bandwidth, and the client's computational requirements. While edge computation is already a fact, its consistency, resiliency, and reliability are not guaranteed.

Our proposed model is general enough for many types of applications, and yet is feasible for actual implementation and deployment. Moreover, our initial implementation shows that our design can be easily deployed using industry standard runtime engines on commodity hardware.

REFERENCES

[1] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha. Real-time video analytics: The killer app for edge computing. *Computer*, 50(10):58–67, 2017.
[2] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. In *SOSP*, 1995.
[3] J. Corbet. Checkpoint/restart in userspace. LWN https://lwn.net/Articles/572125/, 2013.
[4] V. Cozzolino, A. Y. Ding, and J. Ott. FADES: Fine-grained edge offloading with unikernels. In *HotConNet*, pages 36–41, 2017.
[5] B. Cully, G. Lefebvre, D. T. Meyer, M. Feeley, N. C. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.
[6] Docker checkpoint and restore. https://github.com/docker/cli/blob/master/experimental/checkpoint-restore.md, 2018.
[7] Y. Dong, W. Ye, Y. Jiang, I. Pratt, S. Ma, J. Li, and H. Guan. COLO: coarse-grained lock-stepping virtual machines for non-stop service. In *SOCC*, 2013.
[8] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *VEE*, 2008.
[9] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3), 2002.
[10] B. I. Ismail, E. M. Goortani, M. B. A. Karim, W. M. Tat, S. Setapa, J. Y. Luke, and O. H. Hoe. Evaluation of docker as edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)*, Aug 2015.

[11] OpenFog Consortium Architecture Working Group. OpenFog Reference Architecture for Fog Computing, Feb. 2017. https://www.openfogconsortium.org/ra/.
[12] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. 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.
[13] I. Zhang, T. Denniston, Y. Baskakov, and A. Garthwaite. Optimizing vm checkpointing for restore performance in vmware esxi. In *USENIX Annual Technical Conference*, 2013.