Live Migration of Containers in the Edge

Rohit Das[1] and Subhajit Sidhanta[2]
[1]*Radisys India Pvt. Ltd, Bengaluru, India*
[2]*Dept. of Electrical Eng. and Computer Science, Indian Institute of Technology Bhilai, Raipur, India*

Keywords: Edge Cloud and Fog Computing, Cloud Computing Enabling Technology, Cloud Operations, Cloud Migration, Cluster Management.

Abstract: Being comprised of resource-constrained edge devices, live migration is a necessary feature in edge clusters for migrating the state of an entire machine in case of machine failures and network partitions without disrupting the continued availability of services. While most of the prior work in this area has provided solutions for live migration on clusters comprised of resource-rich servers or fog servers with high computing power, there is a general lack of research on live migration on the low-end ARM devices comprised in edge clusters. To that end, we propose a lightweight algorithm for performing live migration on resource-constrained edge clusters. We provide an open-source implementation of the above algorithm for migrating Linux containers. We demonstrate, using simulations as well as benchmark experiments, that our algorithm outperforms state-of-the-art live migration algorithms on resource-constrained edge clusters with network partitions and device failures.

1 INTRODUCTION

In the edge computing paradigm, part or whole of the computations is performed closer to the source of the data, i.e. in the edge devices itself, which result in a drastic reduction of latency by effectively minimizing the communication delay between the edge devices and the public cloud while increasing the certainty of the results meeting the Service Level Agreement (SLA). Usually most enterprises today run on a hybrid cloud, where a model is trained in a public cloud and passed to a cluster of edge devices that form a private cloud, often referred to as fog. The fog performs some inference tasks based on the above model and the input from the IoT devices, and sends results back to the IoT devices which act as actuators that respond to results.

Mission-critical edge analytical applications have stringent Service Level Agreement (SLA) guarantees which enforce that the application processes a given workloads in near real-time, and is always available. Achieving the Service Level Agreement (SLA) guarantees imposed by the services hosted in a given edge cluster becomes harder when we consider the fact that the edge devices are intrinsically mobile in nature, failure-prone, and work with intermittent network connectivity. In that case, the above mechanisms must be able to adapt to conditions like edge devices leaving and joining the cluster or existing devices changing their location. Some devices will be overburdened with processes taking up a lot of resources, like applications involving deep-learning or neural networks, object-recognition from a live video feed, so uniform distribution of workload among the cluster of devices is essential. In case of failure of a device due to network partition, device failure, or overloading, the container running on the failed device must be migrated before-hand to a standby device, and the new device must be up and running with as little downtime as possible. This mandates as adequately smooth, robust, and lightweight live migration algorithm to manage device failures in edge clusters with frequent overloading and reconfiguration. To that end, this paper makes the following contributions:

- It presents LIMOCE - a novel lightweight algorithm to perform a seamless live migration of a container from one edge device to another one in an edge cluster with minimal overhead.
- We present the design of an open-source middleware that implements LIMOCE.
- Using simulations as well as benchmark experiments, we demonstrate that LIMOCE can provide seamless live migration in a resource-constrained edge cluster as opposed to the state-of-the-art mi-

Das, R. and Sidhanta, S.
Live Migration of Containers in the Edge.
DOI: 10.5220/0011060800003200
In *Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER 2022)*, pages 51-62
ISBN: 978-989-758-570-8; ISSN: 2184-5042

gration algorithms designed for typical cloud environments.

2 RELATED WORK

While live migration is deemed an appropriate solution for systems set up over LAN having bandwidths in the range of 1-40 Gbps, there are some challenges in networks with lower bandwidth. The work on VM hand-off (Ha et al., 2017) focuses mainly on edge clusters connected over a WAN of bandwidth 5 to 25 Mbps. While (Ha et al., 2017) leverages delta migration, their approach does not involve resource-constrained edge clusters.

Redundancy live migration (Govindaraj and Artemenko, 2018) is a significant improvement over the pre-copy, post-copy or the hybrid live migration algorithm by leveraging check-pointing and buffering mechanisms.

Though the algorithm proposed is network, application, or platform agnostic, but it still needs a fixed and minimum bandwidth to send over the container copy, and keep downtime to a specified minimum, as agreed in the Service-Level Agreement (SLA). Also, with ARM devices having low resources, implementing this algorithm will not feasible.

Live Service Migration (Machen et al., 2018) proposes migration algorithms for clusters comprised of mobile devices which resemble a resource-constrained edge device closely, but there is mostly no scaling involved. Also, the Pi cluster will be dealing with faltering and unstable bandwidth, and may or may not be mobile.

Dynamic Provisioning of resources (Urgaonkar et al., 2005) for a resource-constrained cluster deals with situations like flash crowds when a sudden amount of huge traffic overwhelms the cluster by rearranging the topography of the cluster devices to accommodate for the sudden surge.

VM Clusters have been set up to provide cloud services and maintain Service-Level Agreements (SLA) (Voorsluys et al., 2009). The test bed cluster had been setup on resources, unlike single-board Raspberry Pis (RPis). The resource-constrained cluster may often run out of resources or crash, and the network may be unreliable. Hence the need for lightweight containers over VMs, which will be easier on the devices for migration.

(Hajji and Tso, 2016) talks about the performance of a single-device and clustered Raspberry Pis, having a master and worker device setup. They mostly focus on the performance of running Spark and HDFS on the cluster natively as well in Docker containers. However, they mostly depend on the robustness of Spark, while our cluster is intended towards supporting most types of computation over runC [1] containers for ease of robustness and migration. The Glasgow Pi Cloud is one of the notable experiments on setting up Cloud servers using single-board 'toy' computers (Tso et al., 2013). The limitation in the experiment, one that we intend to overcome here is the lack of redundancy to mitigate failure, and lack of research on container live migration in Raspberry Pis. Linux Containers (LXC) have yet to implement checkpoints of network namespaces, which is easily possible using runC.

3 SOLUTION DESIGN

3.1 System Model

Following the standard three-layered architecture of edge computing, we consider a cluster of resource-constrained computing devices comprising of an edge cluster that runs analytical workloads on data collected from a group of IoT devices with the option to offload computation to a public cloud. We assume that the above analytical workloads are run on containers created using state-of-the-art containerization technologies to leverage the associated benefits such as flexibility, ease of deployment, and portability in terms of dependencies. Regarding the composition of the edge cluster, we consider a homogeneous [2] cluster where the devices roughly possess a uniform specification. We also assume that an edge cluster is often prone to intermittent network partitions and communication failures. The devices are also failure-prone with chances of frequent crashes and switch off due to overheating. The workloads that we consider are typically real-time in nature, i.e., require a low response time. They do not usually tolerate large delays and long downtime. The workloads are typically compute and data intensive in nature, i.e., they process a large volume of data and perform relatively intensive computations. Lastly, we assume that there are some devices in the cluster that have higher reliability and have significantly more computing resources than the rest of the devices. Experiments have shown that having a super-peer with at least 4 GB RAM prevent the

[1] Along with CRIU, runC can be used to easily manage, configure and migrate a container, along with its network namespace from one host to another.

[2] This is a standard assumption made in prior research in the field of distributed systems in general. But our solution can theoretically be applied to heterogeneous cluster as well (Urgaonkar et al., 2005).

device from crashing due to overload.

The above group of devices are designated as super-peer devices, while the rest of the devices in the cluster are referred to as worker devices. Hence an edge cluster is comprised of a group of super-peers, referred to as the master cluster, and a worker cluster comprised of the rest of the devices in the edge cluster. The super-peers maintain a list of live worker devices and corresponding standby device. A smaller number of edge devices are designated as standby super-peers to act as redundant devices for the super-peers. In case of failures of a live super-peer, it sends a notification to standby super-peers, which are activated to take the place of the failed live super-peers. We assume that leader-election protocols are applied to elect a new super-peer as soon as possible to reduce delay and downtime. We assume that the live super-peers send the state of the container to the standby super-peers at a regular interval to minimize the delay while migrating the said container.

3.2 Design Principles

To enable the resource-constrained edge cluster to handle data and compute-intensive workloads despite frequent failures, the LIMOCE algorithm supports live migration of failed devices under the following design principles. To enable uninterrupted processing of edge computing workloads, LIMOCE is designed to perform smooth migration of containers running on the failed devices during system crashes and network failures while maintaining downtime as low as possible between detection of failure and subsequent resumption of the service. We consider resource-constrained devices where resources include network bandwidth, and hence the algorithm must be able to work with lower bandwidth for communication amongst devices comprised in the cluster. LIMOCE must be able to enable the cluster to scale up and down seamlessly, responding appropriately to the dynamic nature of the workload and network condition.

LIMOCE is comprised of two tasks - failure handling and overload management. As already mentioned in Section 3.1, an entire edge cluster will be segregated into two parts - the master cluster comprised of super-peers, and the worker cluster comprised on rest of the devices. LIMOCE leverages the delta migration technique proposed in (Ha et al., 2017). We improve delta migration by enabling detection of failure of active nodes using the timestamps of the incremental container state received. When the active node is alive and running, the timestamps between consecutive container states received are used to calculate the average time between receiving each state. This is then used to detect timeout when container data has not been received for a duration exceeding the average interval. Alongside incremental state transfer, we demonstrate that the timeout helps optimize network bandwidth usage, as constant monitoring by the standby node also incurs higher downtime. Instead of using custom tools to create specialised containers to migrate, we leverage CRIU [3] and runC to create highly optimised containers using bare-metal Linux-based tools. This, in turn, makes our solution inherently lightweight and and able to deal with resource constrained situations. Further, by incrementally sending only diff values from the state from source to target instances, we are able to handle bandwidth constrained network connections. In the next section, we discuss how LIMOCE handles device failures by migrating a failed or crashed active device to a standby device.

Table 1: List of Parameters used.

Parameters	Description
hash_list	Hash Table to store state info like availability and associated devices in devices to enable data access in $O(1)$ time.
standby_device_MAC	MAC address of the associated standby device.
last_state	Last incremental container state received from the active device.
Mem_{thresh}	Memory usage threshold, set by the user, usually as a percentage of total memory available.
$Disk_{thresh}$	Disk usage threshold, set by the user, usually as a percentage of total disk space available.

3.3 Failure Handling

In an edge computing environment, any edge device may fail due to random crashes, overloading, or environmental factors such as network failures. With LIMOCE, each worker device will have an associated standby device to fall back on in case of device failure. Following from the delta compressed live migration approach (Svärd et al., 2011), we apply the

[3]Checkpoint/Restore In Userspace (CRIU) is a Linux software that can freeze a running container (or an individual application) and checkpoint its state to disk (Open Source, 2012).

delta compression technique to compress the memory pages and disk content to transfer during live migration. Hence, we refer to our approach as delta migration. Frequent delta migration of data encapsulated by containers among devices at a regular interval will prevent data loss and mitigate downtime. Delta migration not only reduces bandwidth usage but using a suite of compression algorithms, helps reduce VM hand-off time, which in turn, reduces the total downtime as well. According to our algorithm, the master cluster, containing the super-peer devices, as well as the worker cluster, containing the worker devices, will be having a replication factor of $n+k$, n being the cluster size of each type, to maintain redundancy for mitigating unpredictable failures. Further, each worker device will be allocated a standby device from a group of unused devices in the respective cluster. Container information like current memory and disk state shall be sent over after every interval of T seconds after the application of delta migration.

Algorithm 1: Device Activation on Worker device.

```
1:  function ACTIVATE_DEVICE(standby_device_MAC)
2:      ▷ Checks if active device is down
3:      while time.now() < last_seen + T + t do
4:          last_state ← curr_device_state
5:          last_seen ← time.now()
6:          sleep (T)
7:      end while
8:      ▷ The Active device has failed
9:      send msg(MAC,active_device) to curr_master
10:     ▷ Standby device takes over
11:     restart container from last_state_recvd
12:     send container data to the new standby device
13: end function
```

A super-peer from the master cluster will allocate a standby device from the pool and the new device will pick up from where the failed device left off. At the initial step, the algorithm sends the entire container information to the assigned standby device in an asynchronous manner in the background. Following the delta compression approach, the delta difference between the states of each active device and its standby counterpart is sent from the former to the latter in consecutive intervals. That way, the volume of information is minimized, allowing the edge cluster to continue processing compute/data-intensive analytical workloads over an unreliable network with limited overall bandwidth. To avoid the constant overhead involved in probing cluster endpoints in a proactive approach of cluster management, we follow a reactive approach in detecting outages due to network partition or crash of cluster devices. Each designated standby device listens for signals from its corresponding active device to determine whether the latter is active or not within an assigned timeout interval. When a standby device does not receive any information from its counterpart active device after $T+t$ seconds, t being the failure tolerance interval assigned by the user, it signals the active master to take over from the failed active device and marks itself as an active device.

In algorithm 1, which runs on the standby device, lines 3-6 are responsible for checking if a respective active device is down. A timeout is signaled based on the assigned timeout interval and the timestamp of the last state received from the active device. If a timeout does not occur, it simply updates the last seen timestamp, and the last state received. Once a timeout occurs, lines 9-12 illustrate the actions taken by the standby device in informing the current master device of the failure, and subsequently restoring the last state received from the active device. If the master allocates a new standby device, the active device sends delta differences of container state to this new standby device.

Algorithm 2: Device Activation on Master device.

```
1:  function ACTIVATE_DEVICE(standby_device_MAC)
2:      ▷ Marks respective active device as down
3:      active_device ← hash_list [standby_device_MAC]
4:      hash_list [active_device].alive = FALSE
5:      for devices in hash_list do
6:          if hash_list [device].state == "FREE" AND
7:  hash_list [device].alive != FALSE then
8:              hash_list[standby_device_MAC].state =
    "ACTIVE"
9:              standby_device_MAC.activate_device (de-
    vice)
10:             exit
11:         end if
12:     end for
13: end function
```

Algorithm 2 deals with how the current master device will handle the failure of an active device. When it receives information of device failure from a standby device, in lines 3-4, it marks the currently active device as dead. It then proceeds to find a new free device from the pool of devices available, marks the reporting standby device as active, and assigns a free device as the new standby of the newly activated device (lines 5-9).

For a master device, two devices will be on standby and will receive delta differences from the active master. When the active master device fails, a bully leader election (Garcia-Molina, 1982) will decide the new master device, which will then broadcast its own MAC address to the worker cluster. A list

of available devices with their hostname and MAC address will be maintained at every device. It will also contain important information on the devices like their status (dead or alive), if they are an active or standby device, their corresponding standby or active device, etc. When a worker device fails, only the active master can change the status of that worker in its record to mark that the device as inactive. It can choose a new master through leader election and broadcast the newly elected master device after every T' interval. When a master device fails, the new master device will broadcast itself as the new master and make the changes in its record, thus reducing bandwidth usage.

Algorithm 3: Additional device Allocation.

```
1: function ALLOCATE_DEVICE(active_device_MAC,stats)
2:     ▷ Checks if stats are above threshold
3:     if stats.memory > Mem_thresh. OR stats.disk_usage
   > Disk_thresh. then
4:         active_device = NULL
5:         standby_device = NULL
6:         for devices in hash_list do
7:             if hash_list [device].state == "FREE" AND
8: hash_list [device].state.alive == TRUE then
9:                 if active_device == NULL then
10:                    active_device = device
11:                end if
12:                if standby_device == NULL AND de-
   vice != active_device then
13:                    standby_device = device
14:                else
15:                    print("No free device left!")
16:                end if
17:            end if
18:        end for
19:    end if
20: end function
```

3.4 Overload Management

LIMOCE also supports proactive monitoring of the edge devices in the cluster to prevent sudden failure due to overloading. Any under-utilized devices from the cluster can be removed to optimize resource usage. Using YourKit, memory and disk usage can be monitored on each device. When resource usage is above a certain threshold, the active device can request the active master to add more devices to the cluster to distribute the jobs and prevent overloading. If two free devices are found in the pool, the master assigns one as the active device, and the other as a standby device, and adds them both to the cluster. If only one free device is found, the device will be assigned as an active device and added to the cluster. An existing standby device may be reassigned to this active device. If no free device is found, one of the standby devices will be assigned as an active device, and a standby device will be reassigned from the cluster.

In algorithm 3, line 3 checks if the resource usage is surpassing either the memory or disk usage threshold, or both. If so, lines 6-8 look for free devices through the pool, and then in lines 7-13, an active device is added first, since that will be responsible for performing the actual computation. In the next iteration, if another device is found, it is assigned as the standby device, else the user is informed that there is no free device left. Similarly, when resource usage falls below the lower threshold, devices can be unassigned to scale down the cluster. Important information on the eliminated device will be broadcast to all workers before taking it out.

4 CORRECTNESS PROOF

Following the formal semantics used in (Mulyadi and Akkarajitsakul, 2019), we denote our proposed live-migration algorithm LIMOCE as an undirected, connected graph. The vertices in the graph denote edge devices and the graph edges represent communication between devices. We denote the CPU utilization in each edge device by a variable x:u associated with the vertex u in the graph corresponding to the edge device. There is always at least one edge in the graph between the pair of vertices representing one active device and its standby counterpart. One edge represents the communication with the active devices to check whether they are active or going to fail. The other edge is created when the delta differences are sent from the active device to the corresponding standby device in case of failure of the active device. The overload management algorithm is executed in every step at each device in the graph. During the migration of an edge device, it starts execution in the state immediately before it and yields a different state upon execution. Optimality of the algorithm implies that if the sequence of steps is finite then the final step of the computation yields a fixpoint state, i.e., a state that remains unchanged upon execution of any (arbitrarily chosen) migration step.

Our goal is to suggest correctness criteria for formally verifying our proposed live migration algorithm. These correctness criteria guarantee desirable algorithm behavior even if very lax assumptions are made about the behavior of the environment, the topology of the network, and the states of the edge devices. In particular, they allow the environment to

produce or consume loads at any time in an arbitrary manner. Moreover, these correctness criteria are not unduly restrictive, which we demonstrate that our algorithm meets these criteria.

A live migration algorithm is 'correct' if the flow of the algorithm satisfies the following conditions. State Lag: At each instant, the lag between the states of each pair of the active devices and its corresponding standby device is bounded by a fixed state lag Δ, which is a function of the value of the application variables plus the metadata. Load: At each instant, the CPU utilization in each device in the cluster is bounded by a pre-defined threshold. Convergence: At the end of each migration step, all devices in the cluster are in 'sync', i.e., all variables and metadata are in a matching state.

Starting from any state, algorithm execution is guaranteed to terminate in a finite number of steps in a state where the states of an active device and its designated standby are identical. The assignment statement

$$last\_state \leftarrow curr\_device\_state \quad (1)$$

iteratively assigns the state of the active device to a counterpart standby device. Each time the above states are synchronized a timestamp are updated as

$$last\_seen \leftarrow time.now() \quad (2)$$

Hence eventually, the while condition

$$time.now() < last\_seen + T + t \quad (3)$$

evaluates to true resulting in the algorithm to exit the while loop. The core efficacy of live migration involves successfully sending the latest state of one active device to its standby counterpart. Hence we look at the termination condition for executing the while loop in lines 3-7 in Algorithm 1. Since eventually the condition

$$time.now() < last_{seen} + T + td \quad (4)$$

evaluates to false, subsequent Lines 9-12 are executed, which results the state of the standby device to match that of the corresponding active device after execution of the algorithm. Hence, conditions on State Lag, Load, and Converge hold simultaneously, and hence correctness of the algorithm is proved.

5 IMPLEMENTATION

To deal with the possibility of device outage and low-bandwidth and/or faulty network prevalent in edge clusters, the network model has been set up to minimize and optimize communication as much as possible. Since the robustness of the cluster largely depends on successful checkpoint transfer and restoration on the target device, our middle-ware implements an automated check-pointing mechanism to support seamless migration of edge devices.

5.1 Architecture

By design, our algorithm addresses live migration of containers in clusters comprised of low-power single-board computers, e.g. Raspberry Pis. For this paper, we have considered a cluster of Raspberry Pi 4, which consists of a quad-core processor, 2 - 8 GB RAM, with support for both 2.4 and 5 GHz wireless with at least 15 W of power for feasible operations (Raspberry Pi Foundation, 2019). Each device will be running a variant of Linux, i.e. Ubuntu or Raspbian. From our initial proof of concept experiments with a cluster of Pis, we observed that 4 GB RAM is the minimum requirement in each Pi to allow for sufficient memory to run analytical workloads. Each device can be powered either with a compatible charger as per the specifications of the Pis or over Power-over-Ethernet (PoE). In the cluster, during initial setup, one device is designated as the super-peer, which is responsible for setting up active and standby devices from the device pool available. The number of active devices and their corresponding standby devices can be accepted from users as command-line arguments to the super-peer. For the purpose of security, key pairs are generated on each device, and public keys are shared across the entire cluster by the super-peer for maintaining inter-device communication during cluster reconfiguration.

5.2 Network Model

For the purpose of this paper, we are assuming that the network would be failure-prone and the bandwidth is limited, i.e., typically varying between 5 - 50 Mbps. We also assume that the devices will mostly be connected over a wireless connection for easier and faster setup, better mobility, serviceability, and scalability.

The super-peer will initially broadcast an initialization command to designate the rest of the devices as active or standby, i.e., assign some devices as standby to each active device. Consecutively, the standby devices will receive delta differences as heartbeat messages sent from active devices, and keep track of whether the respective device is alive or inactive, and will inform the super-peer in case of any failure based on a timeout. One active device can be connected to multiple standby devices to maintain redundancy to deal with failures, and each standby device may receive heartbeat messages from multiple active devices to migrate the containers being used. The timeout for the migration is calculated from the timestamps of the last heartbeat received from active de-

vices to conserve bandwidth and prevent unnecessary traffic. When an active device goes down, the super-peer parses the heartbeat from all the standby devices attached to it and picks the first reporting standby as the new active device.

5.3 Migration Model

Containers and virtual machines (VMs) are widely used these days to encapsulate a computational unit for flexibility, ease of deployment, and smooth maintenance. Containers hold the edge over VMs being the smaller unit that packages an application along with its dependencies. Following the above trend, LIMOCE performs migration of an application packaged in a container, as opposed to VMs, which are resource-intensive and result in more downtime while migration, as depicted by the result of a comparison on the basis of the usage of bandwidth and RAM during migration in (Machen et al., 2018).

It cannot be used directly by the user but exposes an API that is currently utilized by standard container management tools to facilitate checkpoint and restore. For CRIU to work properly, some features of the Linux kernel must be set, and the kernel needs to be recompiled. The list of key features of CRIU can be found on the CRIU website.

The initial choice for containers was Linux Containers (LXC) (Canonical Ltd., 2008). LXC is a CLI (command-line interface) tool that is widely used given the ease of managing and handling containers with it. The containers are a true encapsulation of a Linux-based system, and they support checkpointing and migration of a running container. However, since the containers are running on devices in an edge cluster, the lack of check-pointing facilities for network namespaces makes it unsuited for our purpose.

Another popular CLI tool for managing containers is runC. runC provides a more granular level of control over the container, such as configuring the startup scripts and processes, setting up network namespaces, and much more. However, runC also has a steeper learning curve compared to LXC. It is not exactly an encapsulation of an entire Linux-based system, but more application-specific. If the underlying computing process needs to be changed, so does the configuration and initial setup.

The advantage runC has over LXC, is that it supports checkpointing for network namespaces, and as a result, containers can be seamlessly migrated between devices. It generates a checkpoint folder, which the user can send to the destination device using file-transfer tools like rsync, which has support for incremental data transfer. The container can then be restored in the target device, and processes resume. For ease of setup, runC containers can be configured using docker images of applications, either pre-existing or user-defined, by simply exporting them. A network namespace is needed to be created for each runC container to be able to connect to the network using the host network namespace. When a container will be migrated, the network namespace will also be migrated to restore connection and computations on the destination device. That way, even if the address of the active host changes, the super-peer can keep track of it and the container network remains preserved.

6 EVALUATION USING SIMULATION

First, we evaluate our migration algorithm using simulation, where we construct different edge computing scenarios comprising standard edge topology using and the algorithm is implemented using a simulation tool.

6.1 Experimental Setup

The above algorithms were implemented using the widely used Mininet simulation framework. The Mininet scripts were run on Mininet VMs on an edge cluster topology constructed using the Python-based API exposed by the popular open-source software-defined networks - OpenFlow SDN.

Network Topography: We set up a simple single-switch network of 10 hosts, with one controller and one switch using OpenFlow. We assigned one host as the master device, 5 as active devices, 3 as standby devices monitoring the active devices, and 1 as a free device. Overall, for each active device in the cluster, we assign two standby devices that maintain replicas of states of the active device.

Device Management: A server script (Mininet script) ran on all the standby devices on IP 0.0.0.0. This is the server where the active devices would send the current state of the computation. On the active device, another script incremented a number (can be passed as an argument by the user) and wrote it to a file to simulate a simple computation. Another script is used to send the file to the standby device using rsync (which needs keys to be exchanged to work) for simulating the transfer of delta differences between states, and also send the hostname and a text to the server running on the standby. Rsync has an option that can be used to send incremental delta differences, suited to our purpose.

On the standby device, we use a script to check for entries sent by an allotted active device to the server, and notify when the active device has not responded within a certain time period (determined by the user) and caused a timeout. This was extended further to inform the master device about the dead active device. The idea was to keep a list of active devices in a file on the standby device, and vice versa on the active device. The entries would include the hostname and private IP of the devices since they are all in the same network. Similarly, on the master device, active, standby, and another list of devices would be managed in files.

Each active device will perform its computation (simple incremental addition) and report its status with the current timestamp. Its corresponding standby devices will monitor its status, and when the active device has not reported its status in a long time, a failure is reported to the master device. The master device then stops all threads corresponding to the failed active device, assigns the first standby device that reported the failure as the newly active device, stops monitoring threads run by it previously, and reassigns standby devices to all the active devices out of the existing ones.

Python API Implementation: We use Python-based API exposed by Mininet to implement different variations in the control flow in an edge cluster to simulate different scenarios for live migration. Since Mininet does not create hosts as a separate process, with its resource constraints, a custom Python code is developed to allot a thread to each host for simulating various actions, such as performing a computation, announcing its live status, reporting failure, etc. There is a master thread that monitors the whole cluster, active threads for active devices, standby threads for standby devices, and a printer thread to print out the current status of each device. A global Python dict is maintained, containing the status, action, associated devices for each device in the cluster, and queues having alive, dead, free devices in them for easier allocation and access.

We also implemented various functions of the master device in the simulation code. The master device can detect failure from its requests, select the failed active device, mark it as dead, select the first standby device that reported the failure, remove it as standby and mark it as the new active device. The master thread was not able to wait for the active threads to stop since it was not the parent thread. The changes in host data is printed on a separate terminal for better debugging.

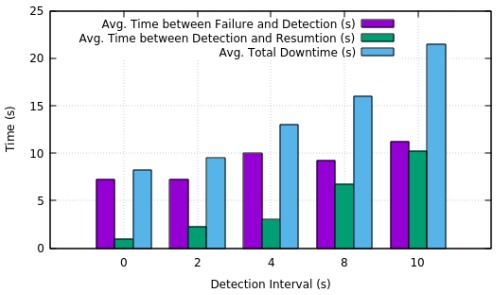
We used threads instead of processes, since processes don't share data among themselves, but make their own separate copy and work on it. Each thread now had its flag, which indicated if a thread was running, and could be used to stop the thread as well. The master thread was able to handle multiple device failures now. A separate thread to consistently print the status of all hosts had been created, which wrote the status to a file, and a bash script showed the latest updated lines.

6.2 Evaluation Methodology

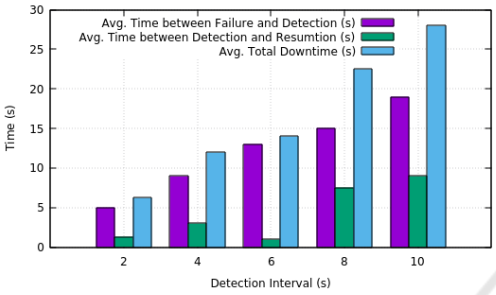
The active devices, along with their hostname, IP, and current state, also reported the current timestamp to their respective standby devices. Any form of communication between any of the devices had a timestamp attached to it, which is logged locally in the devices. These timestamps have been used to calculate the time between each communication from the active device to the corresponding standby device(s), which is used to calculate the timeout on the standby device(s) as well. The difference between the timestamp when a failure of an active device is reported, and when a standby resumes the computation from the last state received has been used to calculate the downtime during a random device failure. Since the timestamps are recorded at the granularity of each communication, the duration of operations like how long it takes for the master device to find a free device and allot it as the new standby, or how long it takes for the standby device to resume operation could also be obtained and observed for further improvement and optimization.

6.3 Simulation Results

We have implemented two variants of the device allocation algorithm (refer to Algorithm 3), and have done a comparative evaluation of them. One variant exactly matches Algorithm 3, while the other one was assigned a dedicated standby device to each active device. In the latter variant, each standby device regularly checks with its corresponding active device to determine if it is alive. However, this approach has the following pitfall. The standby devices transmit broadcast messages to probe the active devices, which results in overhead exceeding 7 seconds in case of loss of response. The above overhead comprises the timeout interval for the broadcast and the delay in detection of outage of an active device from a respective standby device. On the other hand, the overhead with the former approach is the interval between the exchange of broadcast messages from active devices notifying consecutive changes in the status.



(a)



(b)

Figure 1: Detection interval against avg. time between failure, detection ,and resumption, and total downtime for (a) standby devices ping active devices, and (b) active devices announce their live status.

(Ha et al., 2017) illustrates the reduction in VM Handoff time, and consequently, the resultant decrease in downtime, using Delta Migration (Ha et al., 2017). The graphs in Figure 1 demonstrate how downtime changes with respect to change in intervals between consecutive live device detection runs. From close inspection of the figure, we conclude that the second approach performs better.

7 EVALUATION USING BENCH- MARKING EXPERIMENTS

In the benchmarking experiments, we follow the evaluation strategy of prior researchers (Ha et al., 2017) to run synthetic benchmark workloads on a cluster of edge devices emulated by resource-constrained virtual machines from the public cloud.

7.1 Experimental Setup

To benchmark our proposed migration algorithm in a real-world edge computing scenario, synthetic benchmark workloads generated by YCSB (Yahoo Cloud Serving Benchmark) (Cooper et al., 2010) are executed on a cluster of nodes comprised of resource-constrained edge devices has been emulated using virtual machines provided by Amazon Web Services. Each of the host devices are comprised of EC2 (Elastic Compute Cloud) instances of type A1.large instances comprised of 4 GB of RAM and AWS Graviton processors comprising two 64 bit Arm Neoverse cores running Ubuntu 18.04 as the host OS. All the instances are located in the North Virginia (us-east-1) location. For enabling seamless operation of the Checkpoint/Restore-In-Userspace (CRIU) tool used in our implementation, we have loaded the instances with the Linux kernel version 5.10, with relevant configuration options turned on to enable checkpoint-based dumping of process states and network namespaces.

The CLI (Command Line Interface) tool runC has been installed on all the EC2 instances for spawning and running our containers. To emulate real-life edge workloads, YCSB will be run on the instances for generating synthetic workload patterns, with a MongoDB server hosted on one of the instances for processing the above workload. The YCSB client accepts arguments from the console and generates a series of synthetic workloads, comprised of a combination of read/write/update statements, which are executed on the MongoDB backend.

One of the challenges with using the tools runC and CRIU for migration is making sure that the underlying Linux-based kernel of the host is configured and pre-compiled to support check-pointing. Though container managers like LXC and Docker are openly available, LXC does not support network checkpoints, and Docker is not suitable for our purpose due to the experimental state of its check-pointing features. However, the latter can be used to set up the file system for our desired container with all libraries pre-installed. Since runC is a bare-metal container runtime, most of the libraries and networking support do not come pre-installed, but it also allows a lot of fine-grained control over various aspects of the container.

Additionally, to enable communication with external networks, we add routing rules to allow the instances to connect to the default interface eth0 (in this case) or any other interface configured. Both incoming and outgoing packets need to be forwarded to and from the newly created network namespace to the external network via eth0. Since the YCSB client processes only outgoing connections to the underlying MongoDB server, we do not need to add a rule for allowing connections to the MongoDB port. Lastly. we add the path to this new namespace in the config.json file of the corresponding runC container to enable communication.

7.1.1 Running the YCSB Benchmark

The YCSB client is executed on a runC container loaded on a separate active node. The network namespace is configured as described earlier. The client executes workloads A through F according to the alphabetical order of the name of the workload type.

Each active node will be associated with a standby node, which will monitor the status of the active node. An active node will send a packet every 4s to indicate its status to the corresponding standby node since we have observed from empirical results collected from our experiments that this time interval (i.e., 4s) can result in saturating the network as well as keeping the downtime low. Check-pointing is done between each operation of load and run, as well as between execution of different workloads. Incremental data from different checkpoints collected on an active node will be transferred to the corresponding standby node as well. When an active node goes down, its standby will detect the outage through an observed timeout, and restore the container from the latest checkpoint received, and inform the master node as well. For this paper, we assume only soft crashes in case of failures such that some failure resolution tasks can be performed before the active node completely dies and the container is migrated to a standby node. In case of a hard crash, the standby node will have to restart from an older checkpoint, and the requests may not be processed in real-time. Also, rsync may be used to be transfer files in an asynchronous manner, and live status may be sent to the standby node as a separate service.

7.2 Analysis of Benchmarking Results

The benchmarking experiments on the AWS instances were done following the second approach outlined in Section 6.3, where the active nodes send their data to the standby nodes, and the latter node is assigned the responsibility to indicate whether the former is alive. The tool rsync was used to transfer files over the network as its design closely follows that of the delta transfer technique which we follow. While the initial stage of the transfer takes a considerably large time, transfer times of subsequent files are not arbitrarily high. However, to enable rsync file transfer key exchange among all the devices is performed since the algorithm does not assign a fixed device as a master or standby, or active. The downtime observed with varying time intervals of data transfer of 2s, 4s, etc., were similar to the results obtained with Mininet simulation illustrated in Figures 1a and 1b, albeit taking a bit more time due to a real-life network and devices in the cluster.

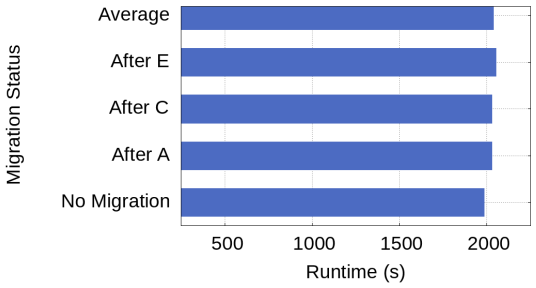


Figure 2: Runtime (in seconds) for the entire workload in various migration scenarios.

Figure 2 illustrates the overall runtime observed with the given YCSB workload sequence for both with and without migration. The bar labeled "No Migration" corresponds to the observation without our proposed algorithm which acts as a control against which we compare the results with migration implemented. As is evident from the graph, with migration, the observed runtimes after workloads A, C, and E are almost comparable with the runtimes observed without migration. Workload E experiences the maximum runtime due to the nature of workload E since it is more resource-intensive than the others.

Figures 4a and 4b depict the variations in observed runtime and throughput for each workload type with and without migration. For Figure 4a, as explained before, workload E uses the maximum resources, and hence the runtime exceeds that observed with the other workload types in all cases. In Figure 3, the time-series variation of throughput for each type of workload is depicted. As can be seen from the results presented, all the migration scenario data closely follow the observations without migration given by the solid line. In all the results illustrated here, we can notice a significant drop in throughput or a sharp increase in runtime whenever migration is being performed. Each migration is performed between a load and run operation for each workload, keeping in consideration how YCSB workloads connect to the MongoDB server and run the workload. However, the drop in performance is marginal compared to the overall picture presented here. We can safely say that even with migration, the performance and overall runtimes of the workloads, as well as the sequence of workloads, remain the same as in a no-migration scenario. This also shows the feasibility of resource-constrained edge clusters in that, even with nodes prone to failure, highly intensive computations can be performed with relative ease and reliability.

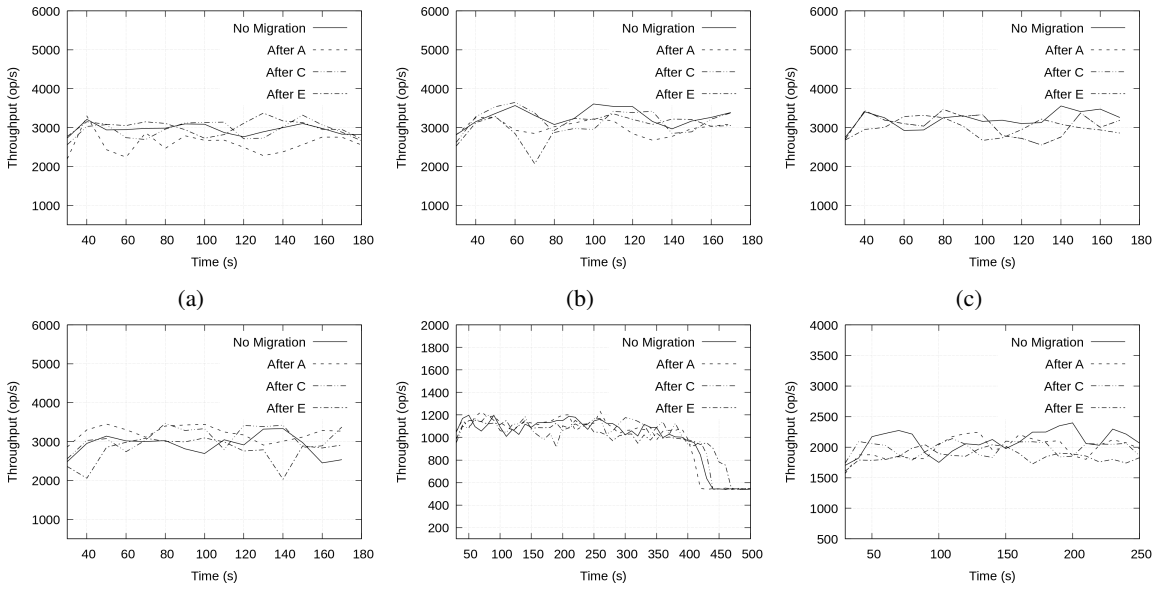


Figure 3: Throughput variation against time for each type of workload.

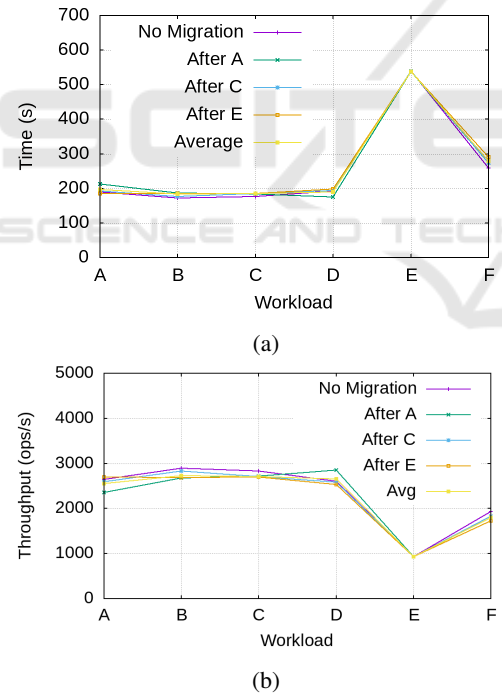


Figure 4: Comparison for (a) runtime (in seconds), and (b) throughput (ops/sec) for various YCSB workloads in various scenarios.

8 CONCLUSION

We have presented a novel algorithm for live migration in network failure-prone clusters comprising resource-constrained edge devices and an open-source implementation of the same.

ACKNOWLEDGEMENTS

This research has received funding under the NetApp Faculty Fellowship scheme from NetApp Inc.

REFERENCES

Canonical Ltd. (2008). Linux Containers. https://linuxcontainers.org.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154.

Garcia-Molina, H. (1982). Elections in a distributed computing system. *IEEE Computer Architecture Letters*, 31(01):48–59.

Govindaraj, K. and Artemenko, A. (2018). Container Live Migration for Latency Critical Industrial Applications on Edge Computing. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 83–90.

Ha, K., Abe, Y., Eiszler, T., Chen, Z., Hu, W., Amos, B., Upadhyaya, R., Pillai, P., and Satyanarayanan, M. (2017). You Can Teach Elephants to Dance: Agile VM Handoff for Edge Computing. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA. Association

for Computing Machinery. https://doi.org/10.1145/3132211.3134453.

Hajji, W. and Tso, P. (2016). Understanding the performance of low power raspberry pi cloud for big data. *Electronics*, 5:29.

Machen, A., Wang, S., Leung, K. K., Ko, B. J., and Salonidis, T. (2018). Live Service Migration in Mobile Edge Clouds. *IEEE Wireless Communications*, 25(1):140–147.

Mulyadi, F. and Akkarajitsakul, K. (2019). *Non-Cooperative and Cooperative Game Approaches for Load Balancing in Distributed Systems*, page 252–257. Association for Computing Machinery, New York, NY, USA.

Open Source (2012). Checkpoint Restore in Userspace. https://www.criu.org/Main_Page.

Raspberry Pi Foundation (2019). Raspberry Pi 4 Tech Specs. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/.

Svärd, P., Hudzia, B., Tordsson, J., and Elmroth, E. (2011). Evaluation of delta compression techniques for efficient live migration of large virtual machines. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 111–120.

Tso, F. P., White, D. R., Jouet, S., Singer, J., and Pezaros, D. P. (2013). The glasgow raspberry pi cloud: A scale model for cloud computing infrastructures. In *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*, pages 108–112.

Urgaonkar, B., Shenoy, P., Chandra, A., and Goyal, P. (2005). Dynamic Provisioning of Multi-tier Internet Applications. In *Second International Conference on Autonomic Computing (ICAC'05)*, pages 217–228.

Voorsluys, W., Broberg, J., Venugopal, S., and Buyya, R. (2009). Cost of virtual machine live migration in clouds: A performance evaluation. In Jaatun, M. G., Zhao, G., and Rong, C., editors, *Cloud Computing*, pages 254–265, Berlin, Heidelberg. Springer Berlin Heidelberg.