Application Agnostic Container Migration and Failover

Martin Terneborg  
Xarepo AB  
Luleå, Sweden  
martin.terneborg@xarepo.com

Johan Karlsson Rönnberg  
Xarepo AB  
Luleå, Sweden  
johan.ronnberg@xarepo.com

Olov Schelén  
Luleå university of technology (LTU)  
Luleå, Sweden  
olov.schelen@ltu.se

Abstract— A key aspect of the cloud is its flexibility and abstraction of the underlying hardware. Historically, virtual machines have been the backbone of the cloud industry, allowing cloud providers to offer virtualized multi-tenant solutions. Today, virtualization by lightweight process containers continue to increase in popularity and make up a larger portion of the cloud, often replacing virtual machines, especially in fog and edge computing. Virtualization can enhance flexibility by enabling support for live migration and failover. Live migration is the process of moving a running instance of a virtual machine or container from one host to another and failover ensures that failures will be automatically detected and the instance restarted, possibly on another host.

This paper presents an overview of current migration techniques, and metrics that can be used for comparison. We also present a proof-of-concept implementation and description of a system that enables support for both live migration and failover for containers by extending current container migration techniques. It is able to offer this to any OCI-compliant container, and could therefore potentially be integrated into current container frameworks. In addition, measurements are provided and used to compare the proof-of-concept implementation to the pre-copy migration technique. We achieve a downtime equal to, and total migration time lower than that of pre-copy migration at the cost of an increased amount of data needed to be transferred.

Index Terms— Container, migration, failover, application agnostic, fog, edge

I. INTRODUCTION

For a long time, virtual machines have been the standard means of providing virtualization in the IT industry. Virtualization enables tasks to exist in isolated environments, which allows for consistent multi-tenancy and portability.

In recent years containers have entered the industry as an alternative to virtual machines. While containers existed previously, they saw a massive increase in popularity in 2013 when the container framework Docker was released [1]. Docker provided features that enabled users to easily create containers, as well as sharing and building upon each other’s container images. Containers are often regarded as lightweight virtual machines, having lower boot-up time and lower resource usage [2] [3]. An important reason is that containers run on the host machine’s kernel, rather than running their own kernel as is the case with virtual machines. This is an advantage for cloud providers and data centers, where each physical machine can run many instances that, efficiently and quickly, can be started or restarted.

In edge and fog computing the drivers and dynamics of lightweight virtualization is taken to a new level, where services should quickly be spun up close to mobile clients that are in desire of services. In such scenarios, there is a need to migrate running services between core and edge (and back again) depending on current demand. Core locations may have higher capacity, be more durable, and be able to serve a larger number of mobile devices independently of geography, while edge and fog locations have a more ideal placement for providing a high quality of service to devices nearby. Besides dynamic migration of service instances, it is also important to support instantaneous automatic failover to ensure service continuity during unforeseen crashes and network disruptions (e.g., when an edge node goes out of operation). Furthermore, container-based virtualization on edge devices, although more efficient than VMs, still suffer from long deployment times [4]. This may be mitigated by migration and failover, as the service’s bootup process does not have to be re-performed when moving the service. The two core concepts in this can be described as follows.

First, migration is the process of moving a running instance of a virtual machine or container across hosts. Typically, migrations are often concerned with being stateful and/or live, meaning that, respectively, the state is preserved after moving the instance and that the moving of the instance is done in such a way that a user of the instance would not be aware that the instance had been moved.

Second, failover allows an instance of a virtual machine or container to automatically be activated on another host upon failure. Unlike migration, failover may not be seamless as failure detection typically cannot be done fast enough to ensure that the end-user would not notice the switch of instance. To obtain seamless failover, typically the application software must be designed as a replicated distributed system, where multiple instances continuously interact. Such design is hard and typically only provided for durable back-end systems. Most front-end services are not designed that way and they would benefit from application agnostic failover at container level.

Being the older and more established of the two technologies, virtual machines typically have support for both...
migration [5] and failover. Containers, however, have little, if any, support for either. In fact, not even Docker, being the most popular container framework, nor Kubernetes, seem to have any support for either [6][7].

In light of this, the scope of this paper is on application agnostic container migration and failover. We provide a general background on the topic, introduce metrics typically used for evaluation and cover alternatives at a conceptual level. We then present our Proof of Concept (PoC) implementation for application agnostic container migration and failover together with an evaluation of the previously introduced metrics.

II. RELATED WORKS

Nadgowda et al. introduce an application agnostic container migration system that is able to migrate both runtime-state and persistent storage [8]. They introduce a federated data layer for migrating the persistent storage and show that the layer induces a low overhead of 1-3% for common workloads. They do not consider failover however.

Stoyanov and Kollingbaum showcases an experimental feature of CRIU [9] that eliminates the process of writing to/reading from disk when saving an application’s state by keeping it in memory [10]. However, the feature does not appear to be stable enough in order to yield any noticeable performance benefits.

Puliafito et al. provides a detailed description of and comparison between some migration techniques [11], but provides no discussion of failover. Although not directly comparable as their experimental setup differs from ours, their results for networks with high throughput and low round-trip time (which corresponds to our experiments), are in a similar magnitude as those that we present in this paper.

Overall we find that there exists a substantial amount of research that considers stateful live container migration, but virtually none that consider failover (whether stateful or not).

III. BACKGROUND

A. Containers and the Open Container Initiative

The ubiquitous notion of containers is that of process trees executing inside any isolated environment.

Today, containers are primarily enabled by a feature of the Linux kernel called namespaces, which allow different aspects of processes to be isolated inside a namespace, restricting the process to only be able to be aware of the information that exists within that namespace or any other namespace nested inside.

Due to a need for industry standards, Docker started a project under the Linux Foundation in 2015 called the Open Container Initiative (OCI) [12]. The project’s goal was to create open industry-standard container formats and runtimes, and has resulted in two specifications, namely the runtime-spec and the image-spec; together, the two are often referred to as the OCI-spec. As only the runtime-spec will be relevant for this paper, the image-spec will not be further explained.

The runtime-spec specifies the requirements of the runtime environment of a compliant container, and a file system bundle that encodes the container that the runtime can perform operations on[13]. The requirements include a state and lifecycle of events of the container, and a number of operations that the runtime must be able to perform on the container. The file system bundle contains two things: a file called config.json that describes metadata about the container, and a root filesystem that should contain the necessary binaries and libraries to run the container.

While the OCI-spec is just a specification, not an implementation, Docker donated its underlying container runtime, runc, to the OCI-project [12] to serve as a reference implementation of the OCI-spec and this, in extension, means that Docker is OCI-compliant.

B. Evaluation metrics

In this paper, three different metrics are used to analyze and compare different migration techniques.

1) Transferred data: defined as the amount of data needed to be transferred between the hosts when migrating. Typically, only the payload data directly related to the state transferred is counted and not protocol-related data.

2) Downtime: defined as the time between when the container on the first host becomes unresponsive and when the container becomes responsive on the second host. This is arguably the most important metric, as it determines the live aspect of the migration.

3) Total migration time: defined as the time between when the migration is initialized and when the container becomes responsive on the second host. As migrations typically are planned, the total migration time is normally less important than the downtime.

C. Migration techniques

Henceforth the hosts and containers that are running the application when the migration is initialized are referred to as the source. The ones that the container will reside on when the migration is finished are referred to as the target. Furthermore, dumping refers to the act of saving a container’s state to disk and a dump is the saved state.

The state in stateful migrations consists of two different types of state, namely runtime state and persistent storage. Runtime state consists of volatile data that is lost when the container exits. Persistent storage consists of data that is not lost on shutdown. The migration techniques described below are only concerned with runtime state since persistent storage is usually handled using shared volume techniques.

The most common migration techniques that exist today are cold-, pre-copy-, post-copy- and hybrid-migration, of which we will describe the former two, as they will be of the most relevance to this paper.

1) Cold migration: This technique is arguably the simplest one, but as such typically also features the biggest drawbacks, namely the longest downtime. The container is simply stopped, and its state is dumped and transferred to the target host. Once the dumped state has been transferred to the target host, the container is restored with the received state. This results in...
the total migration time and the downtime being equally long [11]. Thus the technique is often not desired, as it provides few advantages other than having a low amount of transferred data.

2) Pre-copy migration: The pre-copy migration technique does not transfer the entire state of the container to the target host while the source container is down. Instead, it pre-dumps part of the container’s state (typically its memory pages) when the migration is initialized and transfers the pre-dumped state to the target host, during which the source container is still kept alive and running. As such, the source container’s state may still be modified while the pre-dumped state is being transferred, and thus, in order to avoid any outdated or conflicted state, any of the source container’s state modified during the transfer is marked as modified.

When the pre-dumped state has been transferred to the target host, a final dump of the container is made. This final dump contains any state modified since the previous pre-dump (i.e. any modified memory pages), as well as everything else in the container’s state that is required to restore the container to its latest state. Creating the final dump stops the container on the first host and the final dump is transferred to the target host, where it is used to restore the container.

This technique may also make use of multiple iterations of pre-dumping, such that each iteration dumps and transfers any state modified since the previous pre-dump, with the hope being that the modified state will shrink with each iteration, and thus the size of the final dump will be reduced. Because multiple iterations may be used in pre-copy migration, it is often referred to as iterative migration. While the total migration time may increase in comparison to cold migration, because the entire state, as well as the modified state, has to be transferred, the downtime may be significantly reduced as the size of the modified state is typically much smaller than the entire state [11]. The main drawback of this technique is that the downtime is less deterministic than in the case of cold migration, as the downtime depends on the rate the memory pages are being modified and the amount of data transferred during the pre-copy phase. Also, the amount of transferred data will be greater than in the case of cold migration as not only the entire state has to be transferred, but also any state modified between iterations, making it possibly less suitable for networks with low capacity [11].

IV. PROOF-OF-CONCEPT IMPLEMENTATION

In this section, the features, usage, and some of the inner workings of the implemented proof of concept system (hence referred to as the PoC) are detailed. The source code and full documentation are available in the repository [14]. The PoC is able to offer primarily two features: live migration and failover. It is able to offer this to any OCI compliant container; that is, one that is specified by an OCI file system bundle.

The live migration feature allows a running container to be moved between any remote hosts, whether virtual or physical, that runs the PoC. It does this while aiming to minimize the downtime as well as the total migration time. The migration is stateful and will preserve the latest state of the container.

The failover feature allows a pair of hosts to make sure that, if the source host or the source container itself fails, the target instance will resume the container after detecting a failure at the source instance. Unlike the live migration feature, the failover feature typically does not completely preserve the latest state. This means that once the target host has resumed the container, the state of the container will effectively have been regressed in time. The PoC provides some variables that the user can adjust in order to mitigate the amount of state lost, at the cost of an increased amount of data needed to be transferred between the hosts.

The solution utilizes a technique called iterative migration, not only during migration but also during the runtime of the container. This closely resembles the pre-copy migration technique, described in section III-C2.

A. Major operations and usage scenarios

An installation of the PoC running on a machine will henceforth be referred to as an instance and it may be prefixed with source or target to indicate whether it is being migrated from or to, respectively. A collection of instances serving the same container will be referred to as a cluster. Although the terminology of cluster implies a possibly large amount of instances, at the time of writing the PoC only supports having two instances in a cluster (this is further discussed in section VI).

1) Joining the cluster: The prerequisites for running a functioning cluster of the PoC are two hosts, both of which having an instance of the PoC running and the OCI bundle. In order to start a cluster, the user would start the PoC by providing it with the path to an OCI bundle, and the PoC would then start the container described by the bundle. The second instance could then join the cluster at any time in an ad-hoc manner.

2) Migration: After having properly set up a PoC cluster, including joining with a second PoC instance, a migration request may be issued by telling the source instance to migrate. The source instance will then perform all necessary preparations needed to migrate. After the preparations are done, the source container will have been stopped and the source instance then exits, and the target instance will restore the container using the files it received from the first host.

3) Failover: After having properly set up a PoC cluster, including joining with a second PoC instance, the system will automatically be able to do a failover to the target PoC instance if the first PoC instance fails. The failover scenario resembles that of the migration scenario with two important differences: it is not issued manually and it does not restore from the latest state. As the failover is not issued manually, the target instance needs to automatically detect failures (how it does this is described in section IV-B) and initialize the failover process. After detecting that a failure has occurred, the target instance will restore the container from the latest possible state. This state, however, will not be the latest state
B. Implementation details

1) Data structures: A dump is a collection of files representing part of a container’s (or a process’s) runtime state. The PoC makes use of checkpoint-restore, which is a technique used to save an application’s runtime state to files (checkpointing) and restoring them at a later stage. Originally, one of the use-cases was primarily for saving runtime state during expensive computations, in order to mitigate crashes during runtime, without the developers having to implement functionality for incrementally saving and loading state to/from persistent storage. With the increase of interest in containerization, the use-case has shifted towards migration of containers, and today checkpoint-restore is the backbone of most implementations of container migration techniques.

Checkpoint-restore in userspace (CRIU) is a library for dealing with checkpoint-restore [9]. An important property is that it is completely implemented in user space and not in kernel space, using interfaces to the Linux kernel to provide its checkpoint/restore-functionality. CRIU’s checkpoint/restore-functionality makes it possible to implement not only cold migration, but also pre-copy migration by allowing iterative dumps.

CRIU’s and the PoC’s dumps consist of image files containing different aspects of a process’s state. From the PoC’s point of view, there are three types of dumps: pre-dumps, full-dumps and checkpoint-dumps, where only pre- and full-dumps are covered in this paper.

Pre-dumps enable the iterative migration technique, and contain only the memory pages modified since the previous dump and a link to the previous dump (or all memory pages and no link in case the pre-dump is the first dump made). As such they can not be used on their own to restore a process.

Full-dumps contain memory pages of the process, as well as the other aspects of the process’s state (file descriptors, process tree, etc.). Thus they contain all the information necessary to restore a process to a previous state. However, the full-dumps are linked to previous dumps in the same way that pre-dumps are, i.e. the pages stored are those modified and a link is created to the previous dump.

If a dump B is linked to another dump A, A being B’s parent, the relationship will be denoted as $A \leftarrow B$. The in-memory representation of the dumps consists of the type (pre-, full- or checkpoint) and a number indicating the dump’s chronological order, i.e. the third dump will have number three and so on. This linking of dumps effectively creates a linked list of dumps that can be used to restore a process.

As previously mentioned, if two dumps $A$ and $B$ are linked, $A \leftarrow B$, and $B$ is a full-dump then both $A$ and $B$ are needed in order to restore the process from $B$. This applies to any number of linked dumps as well, which, for the naive approach of simply linking every dump to the previous dump, would present a significant drawback: all dumps ever made would need to be transferred to the target host, in order to restore the container. This would be unfeasible to use with long-running applications, as the number of dumps, and thus the amount of data needed to be transferred would become too large.

The PoC averts this problem by only linking a certain number of consecutive dumps, after which it would not link the next dump to the previous dump, effectively creating a new list. These new and shorter lists are referred to as chains, and the number of dumps in a chain (i.e. the length of the list) is called the chain length. Figure 1 and 2 shows the relationships between the dumps for the naive and the PoC’s approach respectively. Using figure 1 and 2 as an example, in order to restore from dump $d_5$, the naive approach would require all dumps (i.e. $p_0, p_1, d_2, p_3, p_4, d_5$) whereas the PoC’s approach would only require the dumps in $d_5$’s chain (i.e. $p_3, p_4, d_5$). This implies that each chain to restore from must contain all of the container’s state. It should be noted that for the PoC’s approach $p_3$ would contain all memory pages, and as such would likely be larger than $p_3$ in the naive approach.

Henceforth chains that contain all of a container’s state, and thus can be restored from, will be referred to as complete chains and those that do not will be referred to as incomplete chains. Note that for a complete chain $A \leftarrow B \leftarrow C$, $C$ must be a full-dump while $A$ and $B$ are pre-dumps.

As the container runs, the PoC will regularly dump the container at the end of a configurable interval. Whenever a new dump is made, it is linked to the latest dump in the latest chain, unless the configurable maximal chain length has been reached, in which case a new chain is created. The PoC does not hold all previous chains in memory; in fact, it suffices to only hold the two most recent ones. The reason for this will be explained in section IV-B2 and in section IV-B3. Henceforth the most recent chain will be referred to as the current chain and the second to most recent chain will be referred to as the previous chain. Note that the previous chain will always be complete, whereas the current chain will, most of the time, be incomplete.

2) Migration: When the target PoC instance joins the cluster the source instance will transfer both of its chains to the target host. To emphasize, this will not include all previous dumps, only those in the two chains held in memory (i.e. the two most recent chains). In addition, any future dumps created will also be transferred to the other host. This behavior is not specific to migrating, but rather to joining the cluster.
When the user requests a migration the PoC creates another pre-dump, links it to the previous dump in the most recent chain and transfers it to the target host. It then creates a final full-dump (and stops the container) and links it to the previously created pre-dump.

This is almost the same approach as that of pre-copy migration, described in section III-C2, and as such, the migration technique used by the PoC is pre-copy migration. It should however be noted, that the PoC’s behavior slightly differs from that of traditional pre-copy migration. The migration always reuses the last created dump (either pre-dump or full-dump), causing the next pre-dump made during the migration to only contain the modified memory pages.

After creating the final dumps, the source instance will tell the other instance to start restoring the container. Finally the source instance exits and if the migration is successful, the target instance will now be running the container.

Figure 3 shows a timeline of the migration process.

3) Failover: Along with the steps taken when the target instance joins, described in section IV-B2, the source instance also starts sending heartbeat messages to the target instance. It does this regularly at the end of a configurable interval. If the target instance does not receive a heartbeat message for a configurable duration, it will assume that the source instance has failed and initialize the failover process. After the failover process has started, the target instance will try to determine the latest state to restore from by looking for the latest full-dump in the local dump directory. For example, assume the source instance holds the previous chain $A \leftarrow B \leftarrow C$ and the current chain $D \leftarrow E$ in memory at the time it fails and the target instance has not received any other dumps since joining. Also, assume that $A \leftarrow B \leftarrow C$ is completed and that $D \leftarrow E$ is not. Upon failover, the target instance will restore from $C$, rather than $E$ because $D \leftarrow E$ does not contain a restorable state. This is the reason why the source instance must hold the two most recent chains in memory, rather than just the most recent; if the source instance would fail after the target instance joins and before it completes its current chain, the target instance would not have received any completed chains to restore from.

After the target instance has successfully restored the container, it will always set its previous chain to the chain it restored from, discard any dumps made after the latest dump in that chain, and start working on a new chain. This effectively means that the state inside the dumps discarded after restoring would be lost.

As such, the time in between when new chains are created should be limited in order to minimize the amount of state lost on failover. Henceforth this time will be referred to as the chain duration. The PoC provides two variables which one can alter to affect the chain duration: setting the maximum length of the chain and the duration of the intervals in between creating dumps (in fact, the chain duration will always be equal to the product of these two values). Decreasing the chain duration will, however, increase the amount of data needed to be transferred between the two PoC instances. For a more detailed description of the implementation details, we refer to the master’s thesis [15].

V. Results

In this section, the methods used for measuring the PoC’s performance in regards to three metrics, dump size and downtime/total migration time will be presented along with the results. All results were captured on the following bare-metal host system:

- Intel i5-8250U (8) @ 3.400GHz
- 8GB DDR4 SDRAM @ 2400 MHz
- Linux 5.11.2

The PoC instances run inside Docker (version 20.10.5) containers, i.e., the source and target hosts are emulated. For all metrics, different configurations of the chain length ($cl$) and the length of the interval (in seconds) between dumps ($di$) are used.

We disregard the effects of the network capacity and communication as they provide little insight into the PoC itself. We leave studying the PoC with varying network capacities for future work.

For a more detailed description of how the measurements were taken, we refer to the master’s thesis [15].

A. Dump size

The dump size is measured as follows. The PoC is started inside a Docker container and is specified an OCI-bundle to run, and the PoC instance is allowed to run for a set
amount of time, after which the average size of the chains is calculated and divided into pre-dumps and full-dumps respectively. The OCI-bundle used for the measuring is a Redis database (version 6.0.10).

Figures 4 and 5 indicate that the size of the pre-dumps increases linearly in proportion to the number of key-value pairs inserted and that the size of the full-dumps remains constant, independent of the configuration of chain lengths and dump intervals. As such, they seem to indicate that altering the chain length or dump interval has no effect on the dump sizes. The reason for this is likely that the Redis database can be classified as a non-memory-intensive application in that once it has been populated with the key-value pairs, the application does not alter any significant number of its memory pages.

This is also why the size of the full-dumps remains constant: full-dumps are always the last dump in any chain and since no state has changed, neither will their size. It would therefore be of interest to see how measurements differ with applications that write to a wider range of memory pages.

B. Downtime and total migration time

The downtime and total migration time are measured as follows. Two instances of the PoC are started running inside two different Docker containers on the same previously described host machine, thus network delays are minimal and can therefore be neglected. The first instance runs an OCI bundle and the second joins the cluster. As in the case of measuring the dump size, the OCI bundle is that of a Redis database (the same version) into which a set amount of key-value pairs are inserted. The OCI bundle’s command specifies that a script should be run; the script, in turn, starts the Redis database but also starts sending ICMP ping messages to the host machine. An instance of tcpdump [16] is started and assigned to listen for these ICMP messages, and any messages received are logged to a file.

After this setup has been run for a set amount of time, a migration is performed and the downtime is calculated: let \( t_1 \) be the time the last ever message from the source host was received and \( t_2 \) be the time the first ever message from the target host was received, the downtime \( t_{\text{down}} \) is calculated as \( t_{\text{down}} = t_2 - t_1 \). Furthermore, the total migration time \( t_{\text{tot}} \) can be calculated as \( t_{\text{tot}} = t_2 - t_m \), where \( t_m \) is the time the migration request is initialized.

Given the fact that figures 4, 5 and 6 indicate a constant full-dump size, one might expect the PoC to yield a constant downtime, however, figures 7, 8, and 9 show that the downtime increases linearly in proportion to the number of key-value pairs in the database. This discrepancy may be explained by viewing the migration process as two smaller subprocesses:
checkpointing/restoring the process and transferring the state between the hosts. In the case of the PoC, most of the state has already been transferred when the downtime is started and the remaining part is constant (as indicated by the figures). However the time it takes to checkpoint/restore the process will still increase with the size of the container as the number of pages that will have to be written to/read from disk will increase with the size of the container, causing the discrepancy between the full-dump size and the downtime. This would also be the case for pre-copy migration.

1) Pre-copy migration: In order to compare the PoC’s results with current container migration techniques, downtime and total migration time is also measured using pre-copy migration. The reason for using pre-copy migration specifically is that as explained in section IV-B2 the PoC’s migration functionality closely resembles that of pre-copy migration. Furthermore, the PoC can be modified to perform regular pre-copy migration when a migration is requested. All that is required is to disable the continuous dumping of the state, causing the final pre-dump to not be linked to any previous dumps. This would in effect mean that the PoC’s migration feature now could be classified as \textit{iterative migration}. After the continuous dumping has been disabled, the downtime and total migration time can now be measured as previously described (of course, now the chain length and dump interval no longer have any meaning and can be set to any arbitrary values).

Figure 10 shows that the downtime for the PoC roughly equals that of pre-copy migration, as seen in figures 7, 8, and 9, which is expected due to their similarities. In addition, figure 10 indicates that the PoC has a lower total migration time than pre-copy migration, and that the difference between
total migration time and downtime is smaller and constant.
The cause of it being smaller may be explained by the fact
that upon migration the PoC will link the final pre-dump
to the previous pre-dump (i.e. it will contain the modified
memory pages, rather than all) which will already have been
transferred before the migration is started. The reason why it is
constant is likely the same as why altering the chain length
and dump interval does not alter the pre-dump size: the container
does not write frequently enough to a wide range of memory,
causing little state to have changed in the time between the
final pre-dump and full-dump are made.

VI. LIMITATIONS AND FUTURE WORK

As the PoC uses CRIU for checkpointing/restoring it is
limited by CRIU’s capabilities. Currently, there are several
aspects of a process’s state that CRIU can not checkpoint [17].
The common issue is that they rely on something external from
the process itself (e.g. TCP sockets rely on the other end of
the TCP-connection), and thus is outside the control of CRIU
and can therefore not be guaranteed to work.

The current PoC is limited to handle failover only to
one other instance, i.e., cluster size of two. With more than
two instances in the cluster, every instance in the cluster
would initialize the failover process when a failure occurs.
The desired behavior would be that only one of the other
instances would try to recover the container, maintaining a
single and consistent version of the state. This would increase
the complexity of the PoC as the instances would need to
reach a global consensus on which instance should recover
the container and become the primary instance. At this time
they are oblivious to each other’s existence. For this reason
failover with more than two instances has been left for future
work.

As described in section V, we can not conclusively say
how the parameters introduced by the PoC affects the measure-
ments as the underlying application (Redis) does not write
to a wide enough range of memory. We therefore urge future
works on the subject to address this shortcoming.

We would also like to urge future works to investigate the
PoC in the context of fog computing, i.e. when it is run on
multiple hosts which have varying performance, and also when
the capacity of the network is varied.

VII. CONCLUSION

Container migration and failover are new and largely ex-
perimental techniques, often lacking in popular container frame-
works. The work presented in this paper shows that current
container migration concepts can be used and extended to pro-
vide unified and application agnostic migration and failover.

The evaluations were made using our proof-of-concept
implementation. Evaluation results show that one can expect
to achieve a downtime equal to, and total migration time lower
than that of pre-copy migration.

Future works include to perform evaluations with a greater
variety of applications, especially those that write to a wider
range of memory pages.

Whether the proof-of-concept presented in this paper will
be practically deployable remains to be seen, but we hope
that this paper and the provided source code [14], which can
be freely used to verify and extend the results or built into
operational solutions, will be useful in future works regarding
the subject.

REFERENCES

containers : understanding and optimizing the boot
operation”. Theses. Ecole nationale supérieure Mines-
archives-ouvertes.fr/tel-02418752.
1109/CLOUD.2018.00030.
00008.
DOI: 10.1145/2988336.2988337.
https://forums.docker.com/t/container-live-migration/
2285.
com/kubernetes/kubernetes/issues/3949.
Migration”. In: 2017 IEEE 37th International Confer-
ence on Distributed Computing Systems (ICDCS). 2017,
pp. 2137–2142. DOI: 10.1109/ICDCS.2017.91.
live migration of Linux containers”. In: June 2018.
performance evaluation”. In: Sensors (Basel, Switzer-
land) 19 (2019).
[12] OCI - About the Open Container Initiative. 2021. URL:
https://opencontainers.org/about/overview/.
opencontainers/runtime-spec/blob/master/spec.md.
tending current container migration techniques”. MA