Leveraging Linux Containers to Achieve High Availability for Cloud Services

Wubin Li[†], Ali Kanso[†], and Abdelouahed Gherbi[‡]

[†]Ericsson Research, Ericsson, Montréal, Canada
{wubin.li, ali.kanso}@ericsson.com
[‡]Department of Software and IT Engineering,
École de Technologie Supérieure (ÉTS), Montréal, Canada
abdelouahed.gherbi@etsmtl.ca

***Abstract*—In this work, we present a novel approach that leverages Linux containers to achieve High Availability (HA) for cloud applications. A middleware that is comprised of a set of HA agents is defined to compensate the limitations of Linux containers in achieving HA. In our approach we start modeling at the application level, considering the dependencies among application components. We generate the proper scheduling scheme and then deploy the application across containers in the cloud. For each container that hosts critical component(s), we continuously monitor its status and checkpoint its full state, and then react to its failure by restarting locally or failing over to another host where we resume the computing from the most recent state. By using this strategy, all components hosted in a container are preserved without intrusively imposing modification on the application side. Finally, the feasibility of our approach is verified by building a proof-of-concept prototype and a case study of a video streaming application.**

Keywords high availability, virtualization, linux container, cloud computing, middleware.

I. Introduction

The ongoing development and enhancement of cloud technologies [2], [6], [15], [17], [21], [35] have led to their widespread adoption and a rising trend toward hosting applications in clouds. The use of virtual machines (VMs) has been a key enabler of the Infrastructure as a Service (IaaS) cloud provisioning model, where the VMs and their interconnectivity are offered on-demand to the cloud tenants. However, when moving up in the stack to the Platform as a Service (PaaS) provisioning model, hypervisor-based virtualization at the machine level imposes significant overhead in terms of resource consumption, agility, and elasticity, especially for applications with varying workloads demanding elastic deployments [26].

As an alternative to hypervisors, Linux container [6], [17], a pre-cloud existing technology, has emerged as a more suitable solution to serve as an execution environment for the cloud applications deployed at the PaaS level [+]. Containers virtualize the environment at the operating system (OS) level, where each container can be considered as a sandbox in which the application can run in an isolated manner. Applications can be packaged and configured to run in containers and by that, the time and effort needed to scale the deployment can be significantly reduced. In fact, the use of containers is shaping the Linux OS landscape where we see lightweight cloud-native OS such as CoreOS [16] using containers for the addition and management of the applications. Cloud management systems [20] are now also natively supporting the use of containers in the Cloud, even at the IaaS level. Moreover, we recently see major players [18], [24], [36] in virtualization and Linux distributions realizing the potential and perhaps the threat of containers and hence partnering with, and supporting de-facto containerization solutions.

Although deploying applications in virtualized environments simplifies their management in terms of easily scaling up/down the application, it also raises the concerns on the availability of services hosted in clouds. The consolidation of larger number of applications on the same server (which can be virtual) implies a larger impact for the failure of that server. Furthermore, the virtualization layer itself can introduce additional failures. Some effort [5], [27], [33] has been devoted in developing high availability (HA) solutions in hypervisor-based platforms, nevertheless the HA for applications hosted in containerized platforms remains largely unexplored. In this work we aim to fill this gap and demonstrate how we can leverage the use of containers to achieve HA for the services hosted in containerized platforms.

The contributions of this article are threefold: 1) We identify the challenges of providing HA for applications in containerization platforms. 2) We present a novel HA solution that employs HA middleware to compensate the limitation of containerization platforms and achieves high availability for applications in an HA-agnostic manner. 3) We build a proof-of-concept prototype, and verify the feasibility of our solution using a case study of a video streaming application.

The remainder of the article is organized as follows: Section II briefly gives a general background survey on HA and Linux containers. Section III elaborates the design and implementation of our approach. Section IV presents a case-study based evaluation. In Section V we discuss the advantages

[+]Google as a PaaS provider declares launching over 2 billion containers across its global data centers each week [10].

978-1-4799-8218-9/15 $31.00 © 2015 Crown Copyright
DOI 10.1109/IC2E.2015.17

and shortcomings of our approach. Finally, our conclusions and future work are presented in Section VI.

II. Background and Related Work

A. Preliminaries on High Availability

High availability refers to the ability of a service of being functional and accessible with a probability of 99.999% for a given time interval [13]. We believe that achieving HA is mainly based on the following four pillars: 1) *Redundancy*: Software applications and hardware equipment are bound to fail, and hence to eliminate a system's single points of failure, the availability of redundant resources is a must. 2) *Monitoring*: Constant monitoring is essential to detect failure and trigger recoveries. 3) *Checkpoint and backup*: For stateful applications, it is essential to checkpoint the state of the application to enable service continuity after the recovery. 4) *Recovery management*: Automated recovery management is the key to assuring HA. In fact the recovery management, usually implemented as a middleware layer (e.g., OpenSAF [19] and PaceMaker [23]), is what brings the above pillars together to form the complete picture of a HA solution.

Commonly among the existing HA solutions, these four pillars are built by constructing a pool of loosely coupled servers which are self-contained and keep health monitoring (e.g., via heartbeat) each other, whereby applications can failover from one server to others in an event of failure. While HA solutions may significantly differ in implementation, they share the same principle, which is, duplicating critical components via redundancy in an attempt to eliminate single points of failure.

B. HA Solutions: Advantages and Limitations

As illustrated in Figure 1, HA solutions can be implemented with different strategies, resulting in different levels of complexity as well as performance overhead introduced to the application.

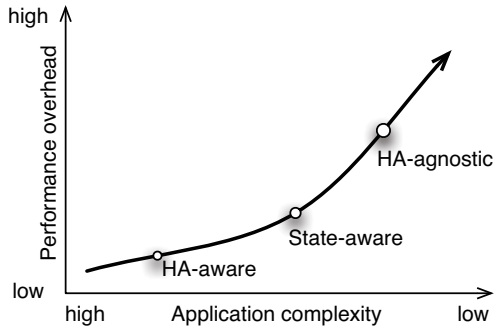

Fig. 1. Application HA integration approaches: application complexity vs. performance overhead.

From the application perspective, an *HA-aware* way is to implement within the application the HA APIs, which makes them part of the application and allows them to directly communicate with the HA middleware to manage the application's availability. This approach has low runtime overhead in terms of performance, however it significantly adds the complexity of developing and configuring the application. To remedy these issues we define in a previous work [14] the notion of *State-aware* applications that are aware of their state, and can checkpoint it through dedicated agents using generic interfaces (such as REST [12]) without being aware of the underlying middleware or implementing HA APIs. We also handle the configuration complexity by automating its generation from high level requirements [30]. However, both the HA-aware and the State-aware approaches are not meant to target legacy stateful applications that are not aware of their state. In this work we tackle the issue of integrating HA for any legacy application without imposing any modifications to its code, which makes the application *HA-agnostic*.

By adding HA-features on top of hypervisors, multiple solutions have been presented for HA in hypervisor-based platforms. For example, VMWare-FT [32] synchronizes the execution of the primary VM and a secondary replica through the *vLockStep* protocol which sends all input data over a dedicated logging link to the secondary copy and replay. The secondary VM as a hot standby of the primary takes over the workload upon failure on the primary, without service disruption. However, this approach requires a fast link network (at least 1 Gbit) between the primary and the secondary and currently only supports VMs with a single logical processor. Instead of using hot standby VMs like [32] and similar to our approach, Chan et al. present an HA solution for cloud servers with snapshot mechanism [3]. A major difference between their solution and ours is the baseline virtualization technology adopted. We believe that Linux containers, as opposed to VMs, have more advantages in terms of performance because of their lightweight-ness. Once again as the increasing adoption of containers, the HA gap in containerized platforms needs to be filled.

C. Linux Containers and HA

Linux container is an OS level virtualization method that allows running multiple isolated Linux systems (containers) on a single operating system host. Unlike hypervisor-based technologies such as VMware, XEN, and KVM, containers do not emulate the underlying hardware layers, but instead provide a virtual environment that has its own CPU, memory, block I/O, and network space etc. using native Linux features such as cgroups, namespace, and chroot. It can be considered as slicing the operating system into smaller units that can host their own applications with their own networking stack and computing resources. Compared with hypervisor-managed VMs, a main advantage of containers is their light weight and management efficiency. The performance investigation of containers is not within the major scope of this work. A comprehensive study on this subject can be found in [11].

The increasing adoption of cloud technologies raises the demand for HA solutions that materialize the four pillars mentioned in Section II-A on cloud platforms. However, a complete HA solution for container-based platforms does not yet exist. In particular, a missing piece is the enabling support for checkpointing the state of a container. There has been

a large amount of effort devoted to process checkpointing, including BLCR [8], DMTCP [1], and ZAP [22]. Nevertheless, these systems were not specifically designed for containers. They either lack features or usually only support a limited set of applications. Moreover, none of them has been part of the mainstream Linux kernel due to the complexity of implementations. At the time of this writing, only OpenVZ [21] has mature and built-in support of this feature and thus selected as the container management platform to demonstrate our prototype in Section IV. A main limitation is that OpenVZ with full features are now only available on 2.6.32 kernel with customization. While it is sufficient for demonstration purpose, we believe this might introduce issues (e.g., security and compatibility) for users when using obsolete kernels.

In order to mitigate the complexity introduced to the kernel side, an ongoing work (i.e., CRIU [4]), is being explored, that is, moving most of the checkpointing complexity out of the kernel and into user space, thus minimizing the amount of the in-kernel changes needed. At the time of this writing, more than 150 patches from CRIU (e.g., memory tracking patches) have been merged into mainline kernel for checkpoint/restore functionality and can be potentially integrated with mainstream container implementations such as LXC [17] and Docker [+] [9]. It should be also noted that the main objective of this work is not to invent a new mechanism of checkpoint/resume a container, but a holistic HA solution on container-based platforms by gluing such enabling features, as well as validating the feasibility of doing so.

III. Proposed Approach: Design and Implementation

Our HA solution is based on deploying the applications inside containers and thereafter constantly checkpointing the state of the container, and by that implicitly backing up the state of the applications inside the container. Our monitoring mechanisms constantly monitor not only the applications, but also the containers, and in case of failure our HA-manager automatically restarts/failovers the container on a healthy node, from the checkpointed state. Nonetheless complementing containers with an HA middleware without considering the application's architecture and the interdependencies of its components may result in a sub-optimal HA solution. Therefore our approach first describes the application with the relevant HA information and based on this information define a placement and recovery policy, and finally HA can be enabled, as shown in Figure 2.

A. The Application Design

Our HA integration starts at the application description where the *System Integrator* describes the applications in an intuitive way. This is done by extending the unified modeling language (UML) component diagram with HA relevant attributes [29]. An example of these extensions is shown in Figure 3, where the application is composed of two components. Comp-A provides Interface-A (which abstracts a given service of functionality). The interface can be defined as stateful or stateless. A component is considered stateful if at least one of its interfaces is. The stateful-ness of the component affects the recovery scope in case of failure. The state sensitivity of the component denotes the maximum duration after which a given state becomes obsolete. This affects the checkpoint intervals required to maintain a healthy state. Comp-A requires the Interface-B provided by Comp-B. This is a functional dependency, governed by the *delayTolerence* attribute that denotes the tolerable delay tolerance in the communication between the two components. This determines how far apart the two components can be placed. The *outageTolerance* attribute determines how long the dependent can survive without its sponsoring component, which also affects the placement strategy to maximize the availability.

[+]With the release of version 0.9 (and thereafter) Docker has replaced LXC with libcontainer [7], an implementation of LXC-like control over cgroups and namespaces, as the default execution environment.

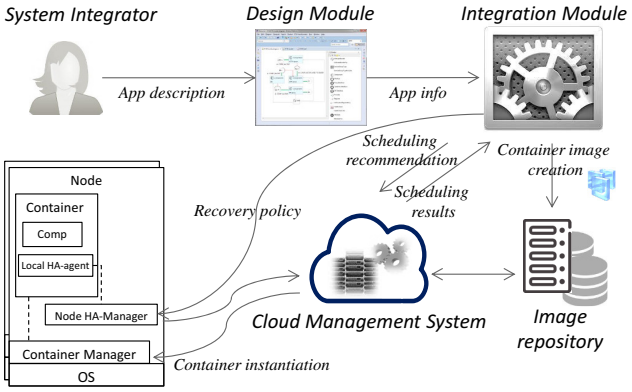

Fig. 2. A holistic view of our approach.

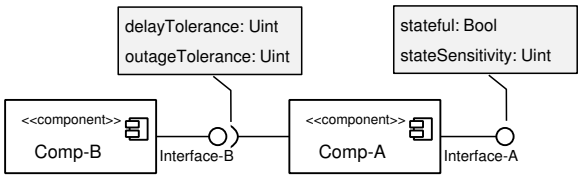

Fig. 3. An example using the extended UML component diagram with HA attributes.

B. Placement and Recovery Policies

The placement and the recovery policies are defined based on the information specified in the design phase. Depending on the delay tolerance of the dependent components, they can either be placed in the same container, same VM, same Server, same Rack, or the same Data-center. The *Integration Module* in our approach processes the application information and then creates the container images containing the application components; thereafter it produces placement recommendations to the *Cloud Management System* on where to place the

containers to maximize their availability. We do not go into further details on HA-scheduling since it is not the main topic of this paper. The recovery policy dictates: the state-extraction frequency, the location where the state is stored, the replication option of the state, and the default recovery for the application failure. When the state is highly sensitive to time, i.e., it changes frequently as time goes by, then the state replication frequency is set accordingly. The state is stored on a distributed and replicated storage in order to ensure that it is accessible by other nodes and resilient to failures. Since the consistency of the state replication can be time consuming, with higher checkpoint frequency the next state is being extracted without overriding the previous state, as shown in Figure 4, whereby if a failure occurs at t_3, the container can be started from the state t_1. Note that this causes additional computational overhead as we will discuss later. Moreover the state compression is only used when it reduces the time cost of writing/reading the original state.

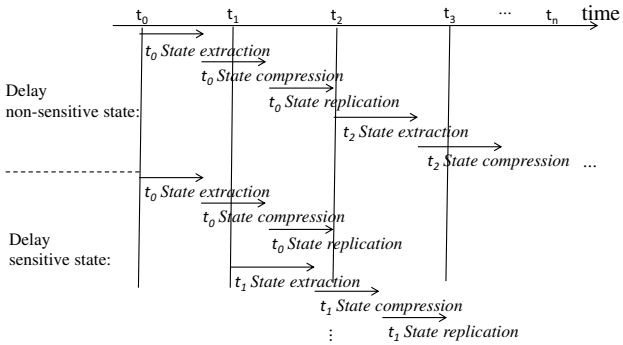

Fig. 4. Checkpointing strategies according to the state sensitivity and delay tolerance.

For the recovery policy it mainly depends on two factors: 1) The stateful-ness of the application: if the application is stateless, then upon its failure it can be restarted within the same container, and resume its functionality. However if the application is stateful then upon failure, the entire container needs to be restarted from its last healthy state. Figure 6 illustrates the recovery strategy to be employed. Note that the restart has precedence over the failover since it can be achieved faster from the locally saved state. 2) The scope of failure: the failure may occur at different scopes, including the application level, the container level, and the node level (a node can be either the VM or the physical machine where the container is hosted). Depending on the scope of failure, the restart may not always be possible, hence the failover becomes inevitable.

C. HA Enabling

In our solution we introduce our own HA-agents, including the Local HA-agents and the Node HA-managers. The objective of adding these components is to complement the four pillars for HA discussed in a previous section. These components perform the monitoring and manage the service recovery based on the recovery policies in case of failures.

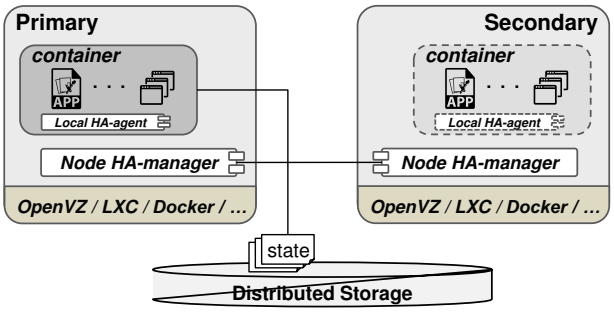

Fig. 5. Architecture overview of the proposed approach.

Figure 5 illustrates the architectural overview of the proposed approach. The Local HA-agent monitors the application inside the container and can control its life cycle (terminate/instantiate). As depicted in Figure 6, for stateless applications, it can react to their failures by trying to restart them internally. However for stateful applications it resorts to the Node HA-manager to restart the entire container of the same node or failover to another node and resume from the last checkpointed state.

The Local HA-agents are complemented by Node HA-managers that have a more global role and visibility to the other Node HA-managers on the different nodes. The Node HA-manager constantly probes the container manager to checkpoint the state of the containers (in a synchronous time intervals) and save it on distributed storage accessible by other nodes. The Node HA-manager also monitors the containers themselves (typically by probing the container manager for the status of the containers). When a failure is detected, an attempt is made to restart the container on the same node, however if the attempt fails, then a failover to another node is triggered. Since the Node HA-manager has a global role, it synchronizes all information with its sibling Node HA-managers during failover. Moreover it is in direct contact with the Local HA-agents located in the containers residing on the same node as the Node HA-manager.

The Node HA-managers, in synchronization with the Cloud Management System illustrated in Figure 2 are also responsible for network management, including network initialization on launching a container, detecting and resolving network conflict issues, and rebuilding the network configuration during failover. In particular, by having a global view of the whole system, they ensure that the IP address of a container is not in conflict with other entities within the same network.

Finally, by leveraging the ability of extracting the state of the entire container, and using our own HA managers to monitor and recover the faulty applications/containers, we achieve HA at the application level in a transparent manner where the application itself is completely agnostic of the HA solution and the application source code is left intact.

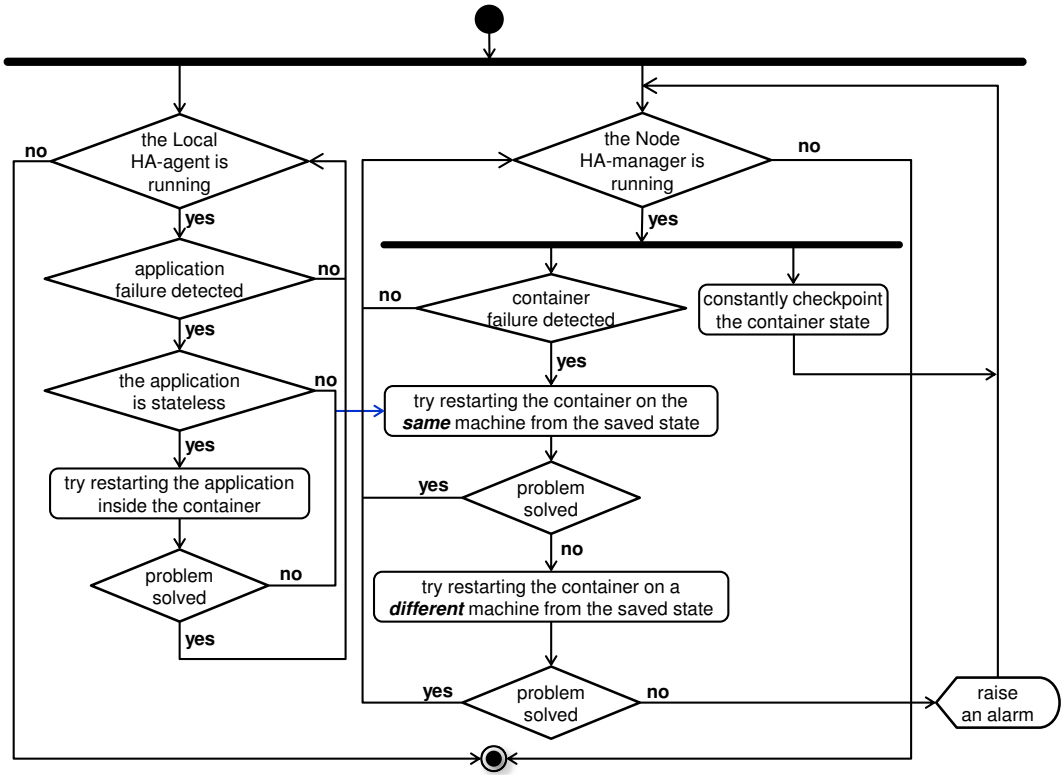

Fig. 6. The flow of detecting failures and recovering.

IV. Case Study and Evaluations

Our extension on UML, as well as the automated configuration mechanism, have been investigated in [14], [29], [30]. In this section, we focus on evaluating the feasibility of the use of Linux container to achieve HA.

A. Case Study Setup

Our case study evaluation is carried out in a virtualized environment which is in line with the architecture illustrated in Figure 5. All the tests are run on a 2.90 GHz Intel Core i7-3520M machine (4 cores) with 8GB of RAM and VMware WorkStation [34] installed. Two VMs (i.e., VM_1 and VM_2 in Table I) are initiated, acting as the role of Primary and Secondary respectively. Each VM is configured with one vCPU, 50 GB of storage, and 2 GB of RAM. The OS installed in VMs is Debian Wheezy. We also use VM_1 as a NFS server to store the state of the container, which is accessible by VM_2. To manage the containers, OpenVZ is installed on top of the OS in both VM_1 and VM_2. At the time of the writing, OpenVZ shows more matureness in terms of the container state extraction (i.e., the snapshotting mechanism) and hence it is more preferable over other container implementations (e.g., Docker and LXC) which only offer partial snapshotting. Nevertheless the choice of the containerization technology does not affect our approach.

In our evaluation, two Ethernet devices are established when a container is created. One is in the host, and the other one is in the container. These two devices are bridged together, composing a virtual Ethernet device. Using this device, the host forwards packets between its physical network interface and the virtual network interface where a container is located, meaning that if a packet is sent to one device it will come out the other device. Whenever a failover from one host to another is triggered, the Node HA-manager re-establishes the same virtual Ethernet interface on the destination host.

TABLE I
Case study settings.

	Role	Operating System
VM_1	Primary host and NFS storage server	Debian Wheezy + OpenVZ
VM_2	Secondary host	

Our case study is a video streaming service using VLC [31], which is initially running in an on-demand mode inside the container hosted on the Primary and is accessible through RTSP [25] protocol. Upon the host failure, Node HA-Manager realizes that it lost contact with its peer and the state is not being updated in the networked file system, which is a clear indication that the host is down. This triggers the container to failover to the Secondary using the latest state checkpointed, without affecting the VLC clients. The CPU usage information on the Primary and the Secondary is continuously collected using *Sysstat* [28] utilities and then exploited to analyze the resource consumption and overhead of our solution. Unless

otherwise specified, each test lasts for 10 minutes, 5 minutes of which is dedicated to stabilizing the whole system. Only the data collected in the second 5 minutes is used. We believe this assures the data acquired have higher and more accurate confidence.

The main objective here is to undertake an assessment of our approach in terms of performance as well as the overhead in terms of CPU resource consumption when using the container and enabling/disabling HA respectively. Currently, we are mainly interested in CPU usage, since the VLC streaming application is neither memory-intensive nor IO-intensive. A comprehensive investigation of performance impact on the application side and in terms of other metrics such as memory and IO is left to future work. More details are discussed in Section V and Section VI.

B. Evaluation Results

We are mainly interested in two aspects: First, the overhead of the HA solution in terms of performance. Second, the correlation between the state size of the container with workload variation.

In our first experiment, we run one VLC server application with varying number of clients (ranging from 0 to 16) under four different settings respectively, i.e., natively on the Primary OS (no HA), using a container as a wrapper but not enabling HA, and using a container and meanwhile enabling HA (with 5 seconds and 30 seconds as the checkpoint intervals). The VLC clients continuously request streaming data during all the tests.

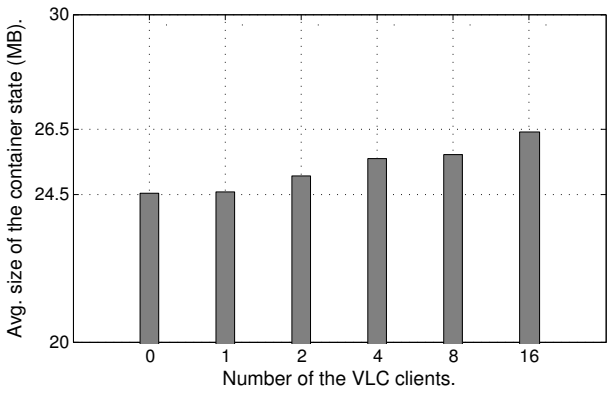

Fig. 7. Container state size vs. number of VLC clients.

Our first observation is that when enabling HA, the size of the container checkpointed increases slightly with the number of VLC clients. This increment is introduced by client sessions. Establishing more sessions results in larger container size. However as we can see from Figure 7, the increment introduced by each VLC client is relatively small. In particular, the size increment is still less than 2 MB even when the number of VLC clients rises from 0 to 16.

Figure 8 demonstrates the CPU usages under four test settings. Following a similar pattern, the average CPU usage on the Primary in each setting rises smoothly as the number of VLC clients increases. This is because the CPU resource

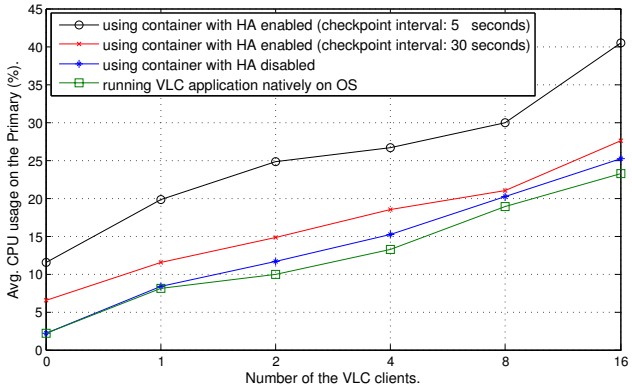

Fig. 8. CPU usages with/without using Container and HA.

consumed to fulfill the requests per client is relatively stable. It is also observed that the overhead of using container is almost negligible (always $< 3\%$) and is not getting heavier as the number of VLC clients increases. When HA is enabled, the CPU usage overhead added to host machine is between 10% (zero clients) to 15% (16 clients). While this may not be very significant overhead on the host machine, it is significant from the perspective of the container usage which can rise from 2% to 12% with zero clients and from 25% to 40% with 16 clients. This overhead is mainly introduced by the continuous checkpointing. In other terms, to increase the efficiency, it is better to overload the container by serving more VLC clients. On the other hand, we also observe that the overhead in terms of CPU consumption depends largely on the checkpoint interval (or checkpoint frequency). As shown in Figure 8, the avg. CPU usage with 30 seconds as the checkpoint interval is very close to the avg. CPU usage without HA. Specifically, their differences are less than 4.5%, indicating that increasing the checkpoint interval from 5 seconds to 30 significantly reduces the overhead of our approach in terms of CPU consumption on the host machine.

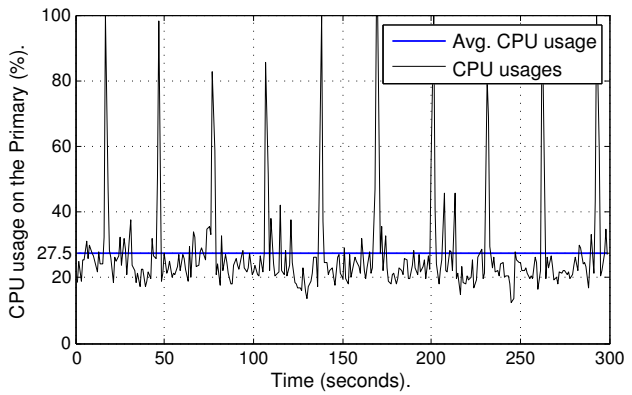

Fig. 9. CPU usages with HA enabled, 1 VLC server running, 16 VLC clients, and 30 seconds as the checkpoint interval.

We analyze this CPU overhead by further examining the scenario with 16 VLC clients and 30 seconds checkpoint

interval. The spikes illustrated in Figure 9 represent the CPU usages when the checkpoint procedure is being executed. It shows that, since each spike ends shortly, the average CPU usage is only 27.5%, even though some of the spike-CPU usages even go up to 100%.

We also investigate the impact of the consolidation of VLC server applications by varying the number of VLC servers running within the container. The number of clients is fixed to 16 and evenly distributed to the running VLC servers (except the case with *zero* VLC server running), e.g., 2 clients are assigned to each VLC server when there are 8 VLC servers running. The checkpoint interval in this experiment is 5 seconds.

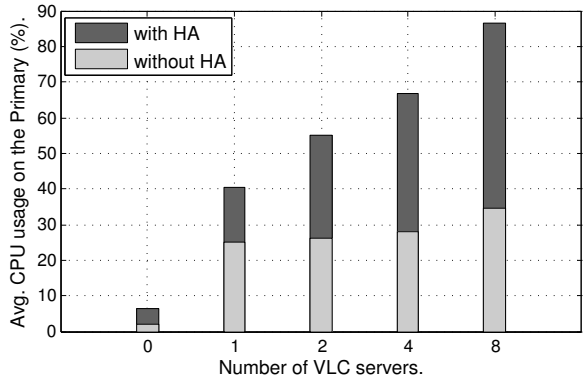

Fig. 10. Avg. CPU usage: the impact of VLC server consolidation, with 16 VLC clients, and 5 seconds as the checkpoint interval.

TABLE II
THE IMPACT OF VLC SERVER CONSOLIDATION.

No. of VLC servers	Avg. CPU usage on the Primary		State size (MB)	Recovery time (s)
	with HA	without HA		
0	6.32%	2.06%	4.418	0.17
1	40.43%	25.16%	26.418	0.55
2	54.94%	26.04%	48.384	0.79
4	66.77%	28.16%	93.761	0.98
8	86.60%	34.49%	192.238	1.16

By examining Table II and Figure 10, an interesting finding is that the average CPU usage without HA does not vary largely with the number of VLC servers. The reason behind this is that the number of clients is constant (i.e., 16) and as aforementioned the amount CPU resource to serve a client request is relatively stable. In contrast, when enabling HA, the overhead increases clearly as the number of VLC servers increases. However, as we already mentioned previously, this overhead can be mitigated by reducing the checkpoint frequency accordingly.

In addition, the checkpoint size of the container also dramatically increases with the number of VLC servers. Intuitively, the size of the container is proportional to the size of the memory required to execute the VLC server applications hosted. Without rigorous mathematical proofs, a venturing guess from Table II could be that the size of the container scales linearly with the number of VLC servers running inside. This can be given by the following expression:

$$S(n) = S_{base} + n * S_{app}$$

where n is the number of the VLC servers, S_{base} is size of the container itself, and S_{app} is the size of a VLC server. In our experiments, we can approximate the values as following through *linear fitting*: $S_{base} \approx 2.50$ MB, $S_{app} \approx 23.51$ MB.

Finally, an evident and important observation from Table II is that the time consumed on failover recovery is spectacularly short ($550\ ms$ when running one VLC server). Given the fact that on the destination host the saved state of the container needs to be loaded into memory and the network needs to be rebuilt, we can conclude that the HA of the VLC server application is achieved effectively and efficiently.

V. DISCUSSIONS

As aforementioned, zero-overhead HA solution does not exist. When adopting checkpoint/resume strategy to achieve HA, performance overhead imposed on the application side is inevitable. In essence, this performance impact is mainly dependent on three factors, i.e., the efficiency of checkpoint procedure, the time interval of checkpoint (i.e., checkpoint frequency), and the size of the container. As elaborated in the previous section, increasing the checkpoint interval can significantly reduce the HA overhead. Another possible approach to mitigate the HA overhead is *incremental checkpoint*, which checkpoints the *delta* since the latest checkpoint instead of the whole container state and then merge the delta with the previous ones to construct a full state. However, a main cost of this mechanism is that the system needs to constantly keep track of the change of a container, which might also introduce (possibly even larger) overhead.

In summary, if the application is not very state sensitive, then the HA overhead can be insignificant. In our case study, the clients are receiving the video on demand service from the streaming VLC server, hence if a failure occurs, the VLC client session is reserved, and therefore the user will have to watch again the last portion of the video equivalent to the checkpoint interval at a worst case scenario. In other terms for a 2.5% CPU overhead with a 30 seconds checkpoint interval we can achieve an acceptable user experience. However this may not be suitable for state sensitive applications, where the state has to be checkpointed every few milliseconds. Moreover, our solution does not protect against errors in the execution environment (the container in our case). Since the entire state of the container is replicated, the error is likely to be carried out to the secondary machine. Using an intrusive approach for HA where the application is responsible for checkpointing its own state (as in OpenSAF [19] and PaceMaker [23]), may protect against the errors in the execution environment. However, apart from the additional complexity on the application side, the network connectivity including the sessions will be lost, since this is state information outside the reach of the application. Our approach handles this issue and at the same time leaves the applications intact.

VI. Concluding Remarks and Future Work

In this work, we present a novel approach using Linux containers and middleware technologies to achieve HA for cloud applications. The middleware is comprised of a set of agents, which are defined to compensate the limitation of Linux containers in achieving HA. By encapsulating the application inside Linux containers, our approach is transparent to the applications, indicating its particular suitability for incorporating high availability to legacy software applications that were built with no HA capabilities.

We also investigate the overhead of the proposed approach using a video streaming case study, demonstrating that our approach has an acceptable overhead which can be tuned by adjusting the checkpoint frequency. It also shows that upon failures, the application recovery time even in case of failover is negligible. To the best of our knowledge, this is the first work that addresses HA for cloud applications by integrating technologies of middleware and Linux containers.

In addition to what have been mentioned in the previous sections, future directions of this work also include a comprehensive performance evaluation of the proposed approach on different containerized platforms for a wider range of applications with different workloads characteristics.

Acknowledgments

This work is partially supported by Ericsson Research and Mitacs Accelerate program. We are grateful to the CRIU team (especially Kir Kolyshkin and Andrew Vagin) for providing valuable information about OpenVZ and CRIU. We also thank Maxime Turenne for sharing his knowledge and experience on HA middleware. Finally, we would like to thank the anonymous reviewers for their constructive feedback.

References

[1] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009)*, pages 1–12. IEEE, 2009.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[3] Hoi Chan and Trieu Chieu. An approach to High Availability for Cloud Servers with Snapshot Mechanism. In *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, page 6. ACM, 2012.

[4] CRIU. Checkpoint/Restore In Userspace. http://criu.org/, visited October 2014.

[5] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174, 2008.

[6] Docker. Docker: The Linux Container Engine. http://www.docker.io, visited October 2014.

[7] Docker. libcontainer source. https://github.com/docker/libcontainer, visited October 2014.

[8] Jason Duell. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. *Lawrence Berkeley National Laboratory*, 2005.

[9] Pavel Emelianov and Serge Hallyn. State of CRIU and Integration with LXC. Presented at Linux Plumbers Conference 2013.

[10] Eric A. Brewer. An Update on Container Support on Google Cloud Platform. http://googlecloudplatform.blogspot.ca/2014/06/an-update-on-container-support-on-google-cloud-platform.html, visited October 2014.

[11] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. *IBM Research Report*, 28:32, 2014. July 21, 2014.

[12] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[13] Jim Gray and Daniel P. Siewiorek. High-Availability Computer Systems. *Computer*, 24(9):39–48, 1991.

[14] Ali Kanso and Yves Lemieux. Achieving High Availability at the Application Level in the Cloud. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, pages 778–785. IEEE Computer Society, 2013.

[15] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[16] LXC. CoreOS - Linux for Massive Server Deployments. https://coreos.com/, visited October 2014.

[17] LXC. LXC - LinuX Container. https://linuxcontainers.org/, visited October 2014.

[18] Microsoft News Center. Docker and Microsoft Partner to Bring Container Applications across Platforms. http://news.microsoft.com/2014/10/15/dockerpr/, visited October 2014.

[19] OpenSAF. The Open Service Availability Framework. http://opensaf.org/, visited October 2014.

[20] OpenStack. Docker in OpenStack. https://wiki.openstack.org/wiki/Docker/, visited October 2014.

[21] OpenVZ Project. OpenVZ Linux Containers. http://openvz.org, visited October 2014.

[22] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A system for Migrating Computing Environments. *ACM SIGOPS Operating Systems Review*, 36(SI):361–376, 2002.

[23] Pacemaker Team. Pacemaker: A Scalable High Avaialbility Cluster Resource Manager. http://clusterlabs.org/, visited October 2014.

[24] Redhat Press Release. Red Hat and dotCloud Collaborate on Docker to Bring Next Generation Linux Container Enhancements to OpenShift Platform-as-a-Service. http://goo.gl/OMwkSl, visited October 2014.

[25] Henning Schulzrinne. Real time streaming protocol (RTSP), 1998. https://tools.ietf.org/html/rfc2326, visited October 2014.

[26] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: a Scalable, High-performance Alternative to Hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287, 2007.

[27] Stratus Technologies. Marathon everRun MX. http://www.stratus.com/Products/Software/everRun/everRun-MX, visited October 2014.

[28] Sysstat. Sysstat Utilities. http://sebastien.godard.pagesperso-orange.fr, visited October, 2014.

[29] Maxime Turenne, Ali Kanso, Abdelouahed Gherbi, and Ronan Barrett. Automatic Generation of Description Files for Highly Available Services. In *Software Engineering for Resilient Systems*, pages 40–54. Springer, 2014.

[30] Maxime Turenne, Ali Kanso, Abdelouahed Gherbi, and Samer Razzook. A Tool Chain for Generating the Description Files of Highly Available Software. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, pages 867–870, 2014.

[31] VideoLan Organization. VLC Project. http://www.videolan.org/vlc/index.html, visited October 2014.

[32] VMware Inc. VMware Fault Tolerance. http://www.vmware.com/files/pdf/VMware-Fault-Tolerance-FT-DS-EN.pdf. visited October 2014.

[33] VMware Inc. VMware High Availability Concepts, Implementation and Best Practices. http://www.vmware.com/files/pdf/VMwareHA_twp.pdf. visited October 2014.

[34] VMware Inc. VMware Workstation. http://www.vmware.com/products/workstation, visited October 2014.

[35] VMware Inc. vSphere ESX and ESXi Info Center. http://www.vmware.com/products/esxi-and-esx/overview.html, visited October 2014.

[36] ZDNet.COM. VMware Buys into Docker Containers. http://www.zdnet.com/vmware-buys-into-docker-containers-7000032947/, visited October 2014.