Failover Time Evaluation Between Checkpoint Services in Multi-Tier Stateful Applications

Demis Gomes, Glauco Gonçalves, Moisés Bezerra, Djamel Sadok
Federal University of Pernambuco
Recife, Brazil
Email: {demis.gomes, glauco, moises, jamel}@gprt.ufpe.br

Patricia Takako Endo
University of Pernambuco
Caruaru, Brazil
Email: patricia.endo@upe.br

Calin Curescu
Ericsson Research
Kista, Sweden
Email: calin.curescu@ericsson.com

***Abstract*—Cloud applications are offered to users with high availability and minimal data loss. Any (hardware or software) failure must be detected and recovered quickly, in order to maintain customer trust and avoid financial losses. When we are dealing with multi-tier and stateful applications, the failure recovery process is a big challenge because the whole state of the failed application must be retrieved and restored in a new instance. This process is named as failover; it can be performed by a checkpoint service at application-level or at system-level. Depending on the location of the checkpoint data storage, it can be classified as non-collocated, collocated warm, or collocated hot. This work presents an evaluation between these two checkpoint services in both virtualized and physical environments, considering a multi-tier and stateful application.**

I. Introduction

Platform-as-a-Service (PaaS) is a Cloud Computing model that provides mechanisms to manage, deploy, and execute applications over the Internet [1]. In order to maintain a high availability of services to their customers, a PaaS provider must recognize failures and react as soon as possible, maintaining the application working with minimal data losses. This challenge becomes more complex when considering multi-tier stateful applications, once that each tier handles with different execution states. In this way, a failover process, which involves restoring the application in another instance, should be implemented in order to make checkpoints, periodically preserve data at regular intervals [2], and recover this data in another redundant instance [3].

The failover process is composed of three services: checkpoint, error detection, and recovery [3]. The checkpoint service is responsible for storing application states periodically. In this way, when the error detection mechanism is fired, the recovery mechanism will be able to retrieve data that was stored by the checkpoint mechanism. Furthermore, the checkpoint service can be implemented at system-level, in which virtualization technologies provide checkpoint through virtual machine (VM) snapshots; and at application-level, in which the application should be aware and decide which data are more critical, reducing the amount of data to be saved.

Despite the fact that some proposal of the state-of-the-art present an alternative to checkpointing in stateful applications [3], none of them makes a consistent comparison between application-level and system-level checkpoint services. Those works evaluate only their own proposals without comparing them to other approaches. This paper presents an evaluation of checkpoint strategies for stateful applications, comparing checkpointing at system and application levels in relation to failover time. We compared two different scenarios: virtualized, running on top of the KVM hypervisor; and physical, running on physical machines.

II. Checkpoint Service

Checkpointing is a process to save state information necessary to keep the application running at a later time [2]. So, in the case of failure, the application could be restarted from the last checkpoint saved with minimal data loss. The checkpoint data can be replicated among other nodes by the checkpoint service to improve the availability of the information, but when using replication, it is necessary to maintain consistency. The main problem of this service is handling the tradeoff between consistency and resource usage [4]: having more replicas increases availability, but the resource consumption and OPEX (Operating Expense) also grows significantly.

A checkpoint service that works at the application-level requires a state-aware application. Therefore, the application's developer must implement API calls for providing and recovering its state in an integrated mode with other application functions. An HA-agnostic application can be checkpointed and restored by a checkpoint service at the system-level. The checkpoint service periodically creates snapshots of the entire system (VM or container) where the instance is running. A state-aware application also can benefit from a checkpoint at the system-level.

A. Checkpoint service at application-level

According to [5], the architecture is divided into two entities, named Checkpoint Manager and Checkpoint Agents. The Checkpoint Manager stores the state information of each instance that runs an application, receiving information, such as tier (frontend, backend, or database), checkpoint mode (non-collocated, collocated warm, or collocated hot), and the application's name. These information define how checkpoint data will be stored and requested, synchronize data with

978-3-901882-89-0 @2017 IFIP

standbys, and coordinate the failover process. Figure 1 shows the application-level checkpoint mechanism, considering the non-collocated mode.

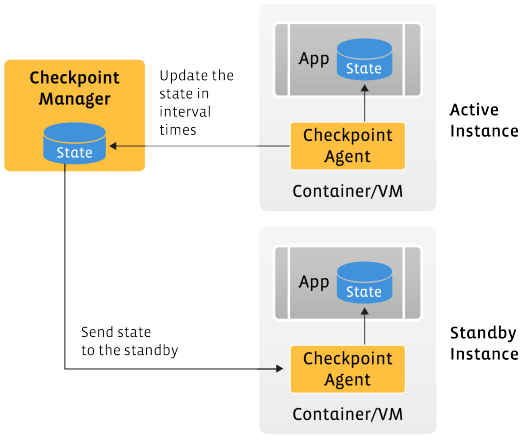

Fig. 1: Checkpoint at application-level with non-collocated mode

The Checkpoint Manager configures the Checkpoint Agent to monitor the application's state in order to able the Agent to update the application's state. The Agent synchronizes the active and standby instances, and restores application data from standby instances. Depending on the checkpoint mode, the state is stored in different locations, as said previously.

The failover process begins with the Checkpoint Manager receiving a fail alert indicating that an application has failed. This way, the Checkpoint Manager seeks the standby redundant replica, referring to a failed instance to make it an active instance, and replacing the failed unit. Afterwards, the Checkpoint Agent, running in a new active instance, changes the configuration from standby to active.

The failover time depends on the checkpoint mode configuration. If the mode is cold or non-collocated, the Checkpoint Agent requests the state from Checkpoint Manager and restores it in the new application instance. If the mode is warm, the Agent recovers the state locally, once that state is already in the standby instance. Finally, if the mode is collocated hot, the application on standby instance already has the last correct state from failed instance, and the Agent only confirms whether this state is updated or not.

B. Checkpoint service at system-level

Similarly to the application-level, the system-level checkpoint architecture [6] has two different roles: primary and secondary nodes. The primary node has a container that runs the active instance of the application, while the secondary has the replica. The Node Agent saves the state of the active instance constantly. This state is shared via NFS (Network File System) with the secondary node and the Node Manager. This Node Manager is the NFS server, and the nodes are NFS clients. Figure 2 describes this approach proposed by [6] with non-collocated mode.

For simplification and naming normalizing reasons, we adapted the system-level checkpoint as follows: the Node Manager is named Checkpoint Manager, the Node Agent is Checkpoint Agent, the primary node is active instance, and the secondary node is standby instance.

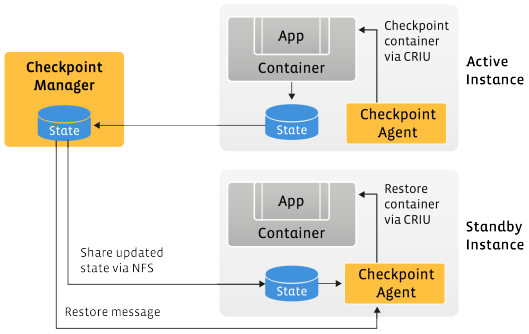

Fig. 2: Checkpoint at system-level with non-collocated mode

For our experiments, we utilize CRIU[1] as system-level checkpoint service. CRIU saves the memory context of a container, allowing it to be restored in another node, if it has a container with the same configuration. In CRIU non-collocated mode, standby instance gets the state via NFS, while in CRIU collocated mode, the state is sent via *rsync*[2]. Basically, the difference between these two approaches is the network delay. With *rsync*, the state is stored on a standby instance directly, while in non-collocated mode, the state is in another machine and is mounted through the NFS.

The failover process occurs when the Checkpoint Manager receives a fail alert. It restores the container in the standby instance with the state shared via NFS, if checkpoint mode is non-collocated, or with the state already in the standby instance shared by *rsync*, if the mode is collocated. When the standby runs the container successfully, this container turns active, which takes the Checkpoint Agent to make checkpoints of the container periodically and sends them to the Checkpoint Manager.

III. Evaluation

Our evaluation is focused on state-aware applications, which can benefit themselves of checkpoint and failover, both at application-level and system-level. We developed a state aware chat application composed of three tiers: frontend, backend, and database. The chat is a stateful application that keeps the state at each tier. The frontend tier stores the chat color, the number of unread messages, and active users. The backend tier keeps room data with messages exchanged between users, and the database tier stores users, rooms, and the relationship between users and rooms.

A. Methodology

We aim to measure and evaluate the failover time under virtualized and physical scenarios, and we are considering

[1]CRIU - https://criu.org
[2]rsync - https://rsync.samba.org/

five checkpoint configurations, as enlightened in Table I. For application-level checkpoint, we implement a service at the application-level (chat) that collects checkpoint data and restores the state (as described in Section II-A), exchanging information through REST messages. States are provided via JSON, a plain text message which can be compressed to reduce data size. On the other hand, we use the CRIU (described in Section II-B) as the system-level checkpoint service for our comparison.

The reason for not evaluating collocated hot in the system-level approach is due to CRIU limitation; the CRIU does not restore a container when it is running, i.e., the container must be stopped and restored when the checkpoint is updated.

The backend of our state-aware chat application contains room data with messages and users logged in. In this case, as the number of messages increases, the room's state size also increases. In this way, the application state size was chosen as factor, with five levels: 1 MB, 2 MB, 5 MB, 10 MB and 25 MB, based on [6]. We defined the length of a message as a tweet (140 characters) composed of *lorem ipsum* message. For each different level and checkpoint mode, experiments were carried out 100 times.

TABLE I: Checkpoint types measured on experiments

	Application-level	**System-level**
Storage	Non-collocated and collocated	Non-collocated and collocated
Checkpoint	Cold, warm and hot	Cold and warm

B. Scenarios

We defined two different scenarios for our comparison: virtualized and physical with the goal to understand the virtualization impact on the checkpoint service. For both scenarios, we configured an entity, named experiment manager, that was responsible for collecting logs, simulating events (such as alerts triggers), and calculating the failover time.

Each checkpoint state size was loaded in the beginning of the experiment, with the active instance recovering the state data in the bootstrap process. We then simulate a fail event, generating a fail message that starts the recovery process.

In the virtualized scenario, two backend instances are executed in LXC on top of two distinct virtual machines hosted in KVM, with the experiment manager running in the host physical machine. On the other hand, in the physical scenario, we configured three physical machines: one for the active backend instance, another for the standby backend instance, and the last for the experiment manager.

C. Results

Regarding to non-collocated mode, in the physical scenario (Figure 3), the application-level presented lower time than the system-level checkpoint in all cases, with the system-level approach being more sensible to the state size increase. On the other hand, in the virtualized scenario (Figure 4), the application-level was more sensitive when the state size increased, presenting the failover time as higher than the system-level when the state size is 25MB. The failover time difference between application-level and system-level was bigger in physical scenario than virtualized one.

Regarding to the collocated modes, in physical scenario (Figure 5), the application-level hot checkpoint obtained the best performance in all cases; and the application-level warm was the worst, presenting the higher increase with variation of the state size. On the other hand, in virtualized scenario (Figure 6), the application-level hot checkpoint and the system-level warm checkpoint had a small variation, being the application-level hot the best.

IV. Discussion

Regarding to the non-collocated mode, the system-level checkpoint presented a smaller failover time in the virtualized scenario than in the physical one. We can explain it, since in the virtualized scenario, the virtual bridge configured in the host machine allows a greater network throughput than in the physical scenario, decreasing the failover time. This experiment also showed that the NFS directly impacts on the failover time when the state size increases; once that in collocated warm mode the failover time in the system-level checkpoint was similar in all cases.

An important result is the similar time between the collocated warm and the non-collocated when using the application-level checkpoint. This result is influenced by smaller state size, because it is a plain-text compressed, which mitigates the state transfer time through the network between manager and agent, just the additional time which makes non-collocated slower than collocated warm mode. Moreover, in application-level, the implementation of an API directly impacts on the recovery time, because the application must recover the state via an API call and confirm that the state was recovered. In system-level with collocated warm, and in application-level with collocated hot, the failover times follow a constant behavior, both in virtualized and physical scenarios, despite the increase of the state size, which shows that state size practically does not influence the failover time in these modes. Despite the closeness of results, collocated hot in application-level obtained smaller failover times than collocated warm in system-level.

Finally, we can state that the application nature and requirements influence in the decision of which checkpoint approach should be used. If an application can be executed with a tolerable data loss, the checkpoint solution can be implemented as an HA-agnostic application, with a smaller time of state saving, operating at system-level. However, if the developer can define which data from the application must be stored, a state-aware application should be developed and a service at application-level must make the checkpoint management.

V. Conclusion and Future Works

In the literature, checkpoint services are proposed in different levels, such as application-level and system-level. However, there is not a consistent comparison between these

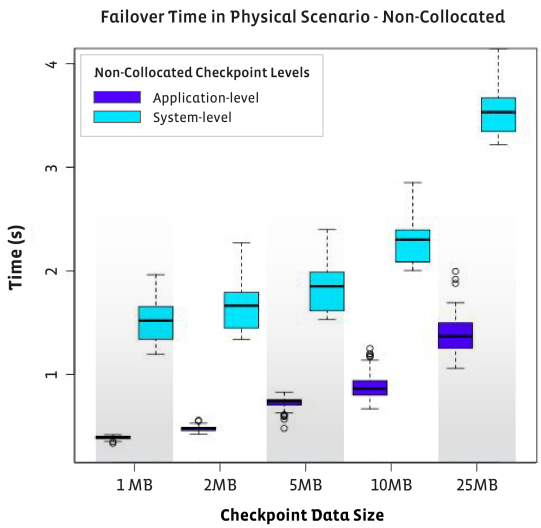

Fig. 3: Non-collocated comparison in physical scenario

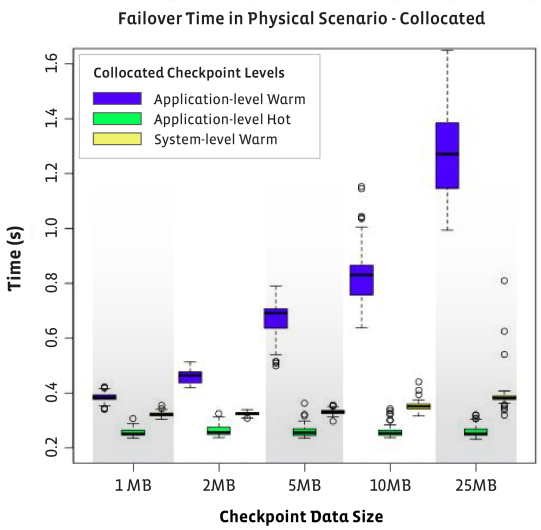

Fig. 5: Collocated comparison in physical scenario

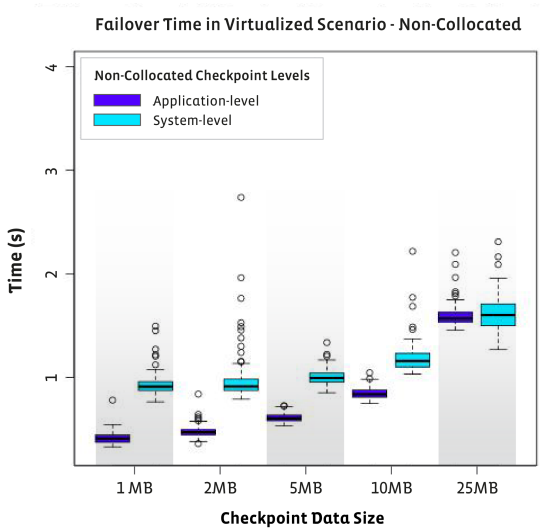

Fig. 4: Non-collocated comparison in virtualized scenario

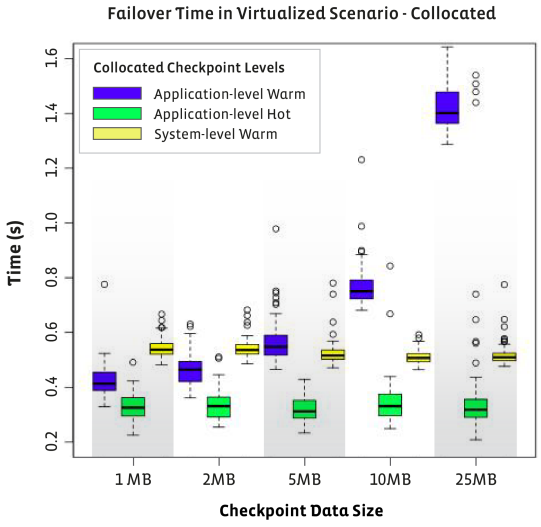

Fig. 6: Collocated comparison in virtualized scenario

services with respect to failover time metric. Our results demonstrated that application-level has smaller times in non-collocated mode considering the physical scenario, and system-level obtained the best results in a virtualized scenario when state size increases, because the interaction between two physical machines does not have shared resources.

As future works, we aim to investigate the load generated on machines in application-level and system-level checkpoint services, as well as checkpoint time between these approaches.

ACKNOWLEDGMENT

This work was supported by the RLAM Innovation Center, Ericsson Telecomunicações S.A., Brazil.

REFERENCES

[1] P. Mell and T. Grance, "The nist definition of cloud computing," 2011.
[2] D. Singh, J. Singh, and A. Chhabra, "High availability of clouds: Failover strategies for cloud computing using integrated checkpointing algorithms," in *Communication Systems and Network Technologies (CSNT), 2012 International Conference on*, pp. 698–703, IEEE, 2012.
[3] M. Nabi, M. Toeroe, and F. Khendek, "Availability in the cloud: State of the art," *Journal of Network and Computer Applications*, vol. 60, pp. 54–67, 2016.
[4] T. Chen, R. Bahsoon, and A.-R. H. Tawil, "Scalable service-oriented replication with flexible consistency guarantee in the cloud," *Information Sciences*, vol. 264, pp. 349–370, 2014.
[5] "OpenSAF Overview – Release 4.4 Programmer's Reference." http://sourceforge.net/projects/opensaf/files/docs/opensaf-documentation-4.4.1.tar.gz/download.
[6] W. Li, A. Kanso, and A. Gherbi, "Leveraging linux containers to achieve high availability for cloud services," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pp. 76–83, IEEE, 2015.