CHECKPOINTING SUPPORT FOR DISTRIBUTED CONTAINERIZED CPS CO-SIMULATIONS

By

ZIQI LI

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

August 13, 2021

Nashville, Tennessee

Approved:

Aniruddha Gokhale, Ph.D (Committee Chair)

Yogesh Barve, Ph.D (Committee member)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Context

Cyber-physical systems (CPS) are computer systems where computation, communication and control are deeply integrated into physical systems, such as in scenarios like smart cities, and transactive energy systems. A CPS involves different heterogeneous systems which need to be analyzed, such as physical architecture, electrical circuits, sensor systems, thermo-dynamics, and networking systems. Each of these aspects is usually simulated by a distinct separate simulation engine. However, CPS simulation demands simulation comprising of integration of these heterogeneous simulation engines, which provides cross-cutting domains' simulation studies. Co-simulation offers principles and techniques for such integration of different individual simulation engines to create a large-scale coupled simulation for CPS study. The IEEE 1516-2010 High Level Architecture (HLA) (Group, 2010) defines the standardized set of services for different parallel running simulators to communicate with each other as well as run in a synchronized manner. In HLA terminology, the simulator is also referred to as a federate. Multiple federates are grouped into a logical entity called a federation.

Although co-simulations have traditionally been hosted in high-performance computing (HPC) clusters, there is a trend to deploy simulation jobs on containers currently, as they provide many benefits. The container techniques, such as openvz and docker(Docker, 2021), are more widely used due to the possibility to save resources and obtain a low overhead compared to a bare cloud server. There is much research work in this new area. For example, in (Y.D.Barve et al., 2020), Barve et al. optimized resource utilization by providing a resource allocation middleware called EXPPO.

However, co-simulations are essentially distributed systems and any distributed system are susceptible to failures arising due to internal application condition as well as due to external environment factors in which they operate. If no fault tolerance mechanisms schemes are employed, it can result in the whole co-simulation has to be restarted completely in the event of failures, which can result in a substantial loss in computing time, energy, and monetary cost to procure the resource. *Checkpoint/Restart* (C/R) offers a potential solution to tackle this issue. Specifically, *checkpoint* involves saving an application's state on stable storage, and *restart* involves restoring the application from the saved states. Periodic checkpointing of an application will allow in the event of failure, to restore the application from the latest checkpoint instead of completely re-running the application from the start. This allows in saving of the computing resources and time for running such applications.

There is much past research work on HPC application checkpointing, such as (Schulz et al., 2004) and (Neves and Fuchs, 1998). The traditional solutions focus on checkpointing applications which are not deployed in the container technologies such as Docker. However, since the applications are now deployed on containers, the containers can fail as well. In these scenarios, the traditional solutions are rendered ineffective. Thus our work focuses on providing checkpointing support for the co-simulations running in a dockerized container running in the cloud computing environments.

1.2 Challenges

However, providing checkpointing support for the co-simulations running in the cloud computing environments is a challenging task.

Firstly, there are accidental challenges involved in providing robust checkpointing schemes for the co-simulations. For instance, the checkpointing schemes may not be able to provide saving of the states of the communication channel used by the co-simulations. For example, assume two federates are communicating by sending messages on the communication channel. If during the checkpointing phase for the two federates, if the message sent by one of the federates is in transit and has not been yet received by the other federates, then one may potentially see a loss in the message packet which may not get checkpointed. Later if one detects any failures in the system and the system decides to rollback both the states of the federates to the previously checkpointed state, the message in transit will potentially be lost if a proper message recovery approach is not utilized. Thus, there is a need for robust underlying coordination and communication mechanism which can address this issue.

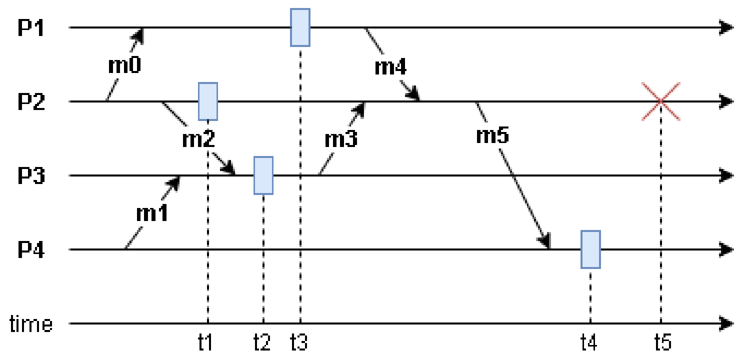

Figure 1.1: Uncoordinated checkpoint scheme for four different communicating processes

Secondly, there are challenges involve in coordinating the checkpointing schemes across the multiple federates of the federation to capture the entire system state at the same instance. Although Docker provides an experimental checkpoint feature that we leverage to checkpoint a single container, it does not provide any coordination mechanism for checkpointing multiple containers at once. A lack of coordination scheme for checkpointing of the distributed states of the system can be an issue for proper restoration of the co-simulation. In Fig 1.1, we see that 4 processes P1, P2, P3, and P4, are communicating with each other. They are checkpointed at t3, t1, t2, and t4, respectively and P2 crashes at t5. After t5, we rollback all of them to the latest checkpoints, so they execute again from the blue squares. After some time, again P2 sends message m5 to P4. However, at this time m5 is not guaranteed to have the same content as before rollback. In the checkpoint snapshot, P4 already received m5, so once it receives m5 again, it may drop the message. In this case, P4 may behave the same as before rollback, while P0, P1, and P2 behave differently at this time. This is where inconsistency occurs since P4 is supposed to behave differently at this time as well. To solve this problem, we must have a coordination mechanism when checkpointing them. In our case, the coordination mechanism should be designed for co-simulation applications running on Docker containers.

Finally, the checkpointing scheme needs to support co-simulation checkpointing in an automated and periodic manner. Furthermore, there is a need for detecting failures or crashes of the docker containers which host the individual federates, such that appropriate recovery of the entire co-simulation can be achieved to the last known checkpoint snapshot. Achieving this is challenging given that as a co-simulation-as-a-service (CaaS) platform for running large-scale simulations we may have a large number of simultaneously running co-simulations which may be spread not just on a single host machine, rather distributed on a set of host machines in a cloud computing cluster. Thus there is a need for automated checkpointing, failure detection, and recovery of co-simulations for a reliable completion of co-simulation execution.

1.3 Solution

To address the above discussed challenges, we make the following concrete contributions in this thesis:

1) **Robust Communication Mechanism:** We modify the underlying communication mechanism for the co-simulation applications, so that it will not cause any problem once checkpoint and rollback.

2) **Coordinated Checkpoint:** We design and implement a coordinated checkpoint mechanism based on the Docker checkpoint feature for co-simulations.

3) **Co-simulation-as-a-service With Checkpoint:** We extend EXPPO (Y.D.Barve et al., 2020), which is a co-simulation-as-a-service middleware, so that the deployed co-simulations can be restored automatically once a failure happens.

1.4 Thesis organization

The rest of the thesis is organized as follows: Chapter II introduces the background and briefly presents other research work on distributed checkpointing and containerized checkpointing. Chapter III presents the system model and solution description. Chapter IV dives into details of our solutions. Chapter V evaluates our work. Chapter VI presents concluding remarks alluding to future directions.

CHAPTER 2

Background and Related Work

2.1 Background

In this section we will cover the background on topics related to the Docker container technology which is used for the co-simulation deployment in the cloud computing platforms, cover topics related to the checkpoint/restore mechanism, co-simulation middleware, literature review in distributed checkpointing.

Docker: Docker is a software for operating-system-level virtualization. (Docker, 2021). Its overhead is very small compared to other virtualization techniques, such as Hypervisor used by virtual machines. It provides a better sandbox environment for security. The Docker file system consists of multiple image layers and a container layer. The image layers store everything of the Docker image, based on which the container is created. The container layer contains all the changes to the file system after the container gets created, such as files creation and deletion. When we commit a running Docker container's file system into a new image, the container layer will be converted into a new image layer upon the old image layers. In this way, we can save the container layer's files of a container. Docker also provides an experimental checkpoint feature to checkpoint the memory of a single running container.

CRIU: Checkpoint/Restore In Userspace (CRIU) is a software on Linux (CRIU, 2021). It allows to saving the working memory state of a running application into the disk. The saved state file can be used later to restore the application. We can use this tool to realize application live migration, snapshots, and many other things. The Docker experimental checkpoint feature is developed based on CRIU.

Portico: Portico is an open-source HLA Run-Time Infrastructure (RTI) implementation, which provides software services for federates to synchronize and send messages to peer federates in a federation (Portico, 2021). The co-simulation applications we use are implemented using Portico. Portico uses Jgroups (Jgroups, 2021) UDP multicast as a communication protocol in a cluster for messaging and discovery. Portico comes with the heartbeat mechanism to detect disgracefully disconnected federates. Each federate sends a heartbeat message to peers periodically. If anyone does not send it before the timeout, others it will be marked crashed and removed from the federation.

Co-simulations: One of the execution patterns followed in a time-stepped-based co-simulation execution is the Bulk Synchronous Parallel (BSP) model (et al., 2000). In this model, every federate has its own computation task and messages with others in each time step. After finishing the work of the current time step, one federate can request for advancing to the next time step. Only after all the federates do so, the

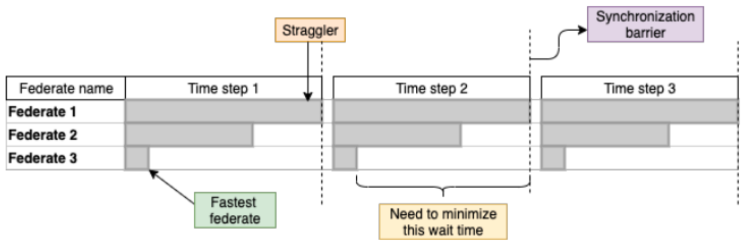

Figure 2.1: An example federation from (Y.D.Barve et al., 2020)

federation's logical time can be advanced. If a federate finishes its work earlier than others, it has to wait rest of the federates to complete execution in a given time step. As shown in Fig 2.1, the federation consists of 3 federates. Federate 1 takes the longest time to execute, and the other two federates have to wait for Federate 1 to finish the work of the current time step before going to the next one. A co-simulation comprises a federation manager and other federates. The responsibility of the federation manager is to manage the federation and provides various services, such as pausing, starting, and stopping the simulation.

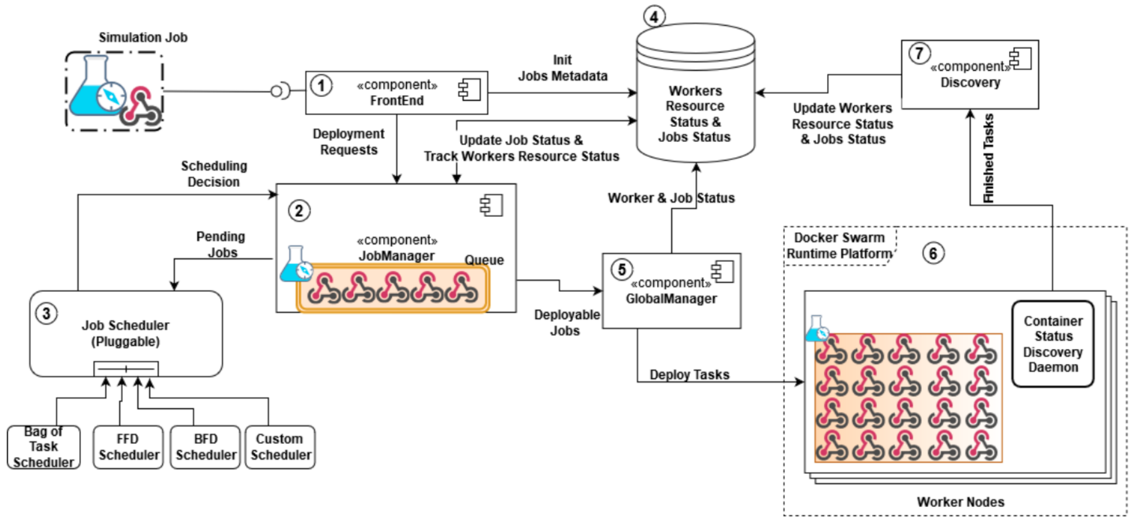

Figure 2.2: EXPPO architecture from (Y.D.Barve et al., 2020)

EXPPO: EXPPO is a co-simulation-as-a-service middleware (Y.D.Barve et al., 2020). Once a user submits a co-simulation job to EXPPO, it deploys the job on Docker containers and executes the job. The user is freed from activities such as configuring the host machine of the co-simulation which are all managed by the EXPPO service. The EXPPO architecture is shown in Figure 2.2. The FrontEnd is responsible for receiving data from Client. The JobManager is responsible for applying scheduling strategy. The GlobalManager is responsible for workers' management using Docker Swarm cluster scheduling. (Y.D.Barve et al., 2020). However, the EXPPO does not provide any kind of checkpointing schemes for the co-simulations, which we are exploring in this thesis.

2.2 Related Work

2.2.1 Distributed Checkpointing

In this section, we present some of the state-of-the-art research work in the area of distributed checkpointing. Two main categories of distributed checkpointing schemes presented in the literature are coordinated checkpointing and uncoordinated checkpointing. The idea of coordinated checkpointing is to coordinate the processes at checkpoint time to ensure that the global state that is saved is consistent. Some of the different approaches are shown as follows:

1) **Application-level Checkpointing:** The application provides global synchronization using a message passing interface (MPI) collective operations (Schulz et al., 2004). One simple example can be is in the iterative codes, wherein the application checkpoint after every N iterations.

2) **Time-based Checkpointing:** In this scheme of checkpointing, each distributed process takes a checkpoint at the same time. In this case, we need to synchronize the clocks of the distributed processes. One solution is offered by (Neves and Fuchs, 1998), in which after checkpointing is completed and the processes wait for some threshold time interval to ensure that all the other processes finish their checkpointing stage. After this threshold, the processes can continue to communicate and exchange messages amongst themselves.

3) **Blocking Checkpointing:** (Tamir and Séquin, 1984) The initiator first broadcasts a checkpoint request to all processes. Upon reception of the request, each process stops executing the application and saves a checkpoint, and sends back an acknowledgement to the initiator. When the initiator receives all the acknowledgements, it broadcasts `ready to proceed` message. Finally, upon reception of the `ready to proceed` message, each of the processes deletes its' old checkpoint and resumes execution of the application.

When the coordinated checkpoint mechanisms checkpoint all the processes at the same time, it may result in resource contention at the file system level during the checkpointing time for large-scale HPC. On the contrary, the uncoordinated checkpointing with message-logging protocols may not exhibit this problem, since it saves checkpoints of each process independently which may happen to be in different time intervals. Recently, the paper (Guermouche et al., 2011) takes the observation that MPI HPC applications send deterministic messages and thus, propose a new uncoordinated checkpoint mechanism.

2.2.2 Containerized Checkpointing

There is also some work on containerized checkpointing. In (H.Zhang et al., 2019), Zhang et al. focused on improving the performance to checkpoint a single container. In (et al., 2020), Ahmed et al. used container Checkpoint/Restart technology to speed up application startup time in the fog computing environment. In (et al., 2019), Karhula et al. used CRIU with Docker to realize live migrating of container state from one internet of things (IoT) device to another.

However, all the above work focuses on checkpointing a single container, and the work on distributed containerized checkpointing is still limited in the literature. In (Berg and Brattl, 2017), Berg et al. used the Docker checkpoint feature to checkpoint an HPC application, specifically the NAS Parallel Benchmarks. However, their result shows that the checkpoint and restore for some applications such as the NPB can be unreliable.

CHAPTER 3

System Model

In EXPPO, GlobalManager manages all the worker nodes and a worker node manages all the federates running on itself. One example of co-simulation deployment is shown in Fig 3.1. GM represents GlobalManager, which manages 2 worker nodes. The federation manager and federate 1 are running on containers on Worker1, while federate 2 and federate3 are running on containers on Worker2. All the containers are controlled and managed by the Docker daemon. To extend EXPPO with checkpoint support, we need to implement 3 basic functionalities on the GlobalManager and worker nodes, which are checkpointing, rollbacking, and detecting failures for a specific co-simulation. The GlobalManager is responsible for deciding when to checkpoint and detect failures. Also, it should be able to send these requests to workers and receive corresponding responses from workers. The worker nodes are responsible for receiving requests from GlobalManager and making use of the commands provided by the Docker daemon to implement these functionalities at a low level.

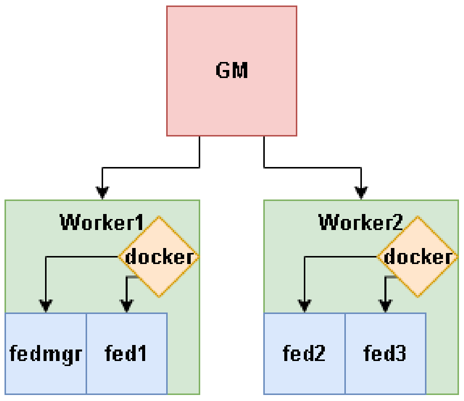

Figure 3.1: Example deployment of co-simulation across two different worker machines. The federates of the co-simulations are shown to be deployed in a distributed manner.

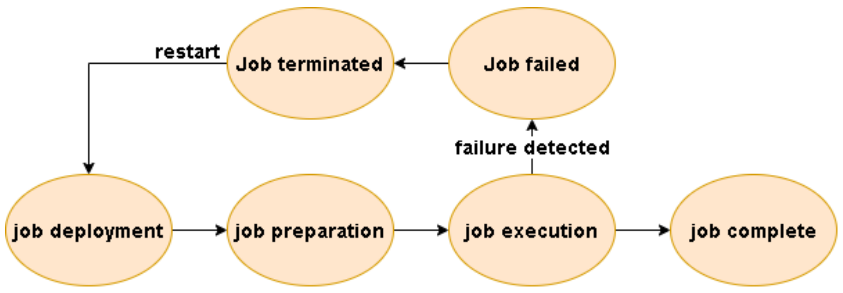

Figure 3.2: Completely restart the job after failure.

During the co-simulation execution time, we assume container failures can happen in each co-simulation

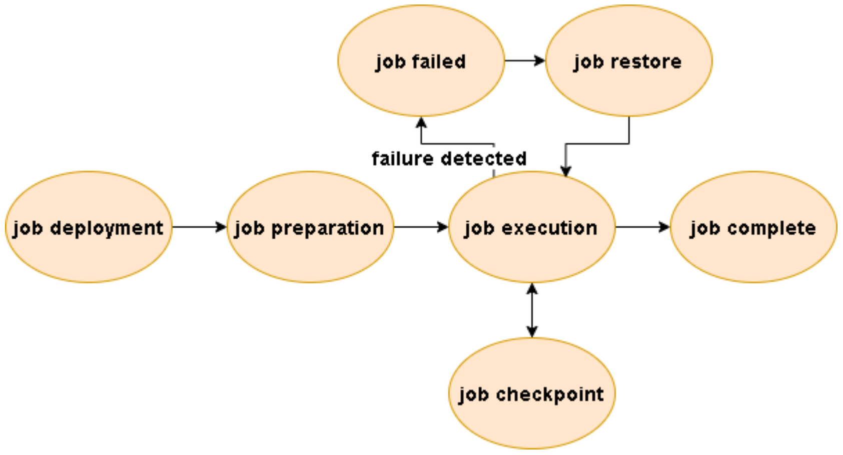

Figure 3.3: Restore the job from a checkpoint snapshot after failure.

job. Any other type of failure, such as application-level failure, does not happen all the time. In this scenario, a naive solution is to completely restart the co-simulation job. As shown in Fig 3.2, The co-simulation first gets deployed by EXPPO, then it needs preparation before execution. While executing, once we detect a failure, we have to terminate the job then deploy it again. However, in this approach, the co-simulation job has to go through the job deployment stage, job preparation stage as well as a part of the job execution stage again, which is very time-consuming. Thus, we want to use the technique of checkpointing and restoring to save time. As shown in Fig 3.3, we periodically checkpoint the co-simulation while executing, once we detect a failure, we restart the co-simulation from the latest checkpoint. In this approach, the failed co-simulation does not need to completely go through everything again, and hence, The time and resources can be saved.

CHAPTER 4

Solutions Details

4.1 Reliable Communication Mechanism for supporting Checkpoint/Restore

Before implementing the checkpointing and restoring solution at the application level, there are some issues in the underlying communication mechanism we need to deal with. In our study, the co-simulation applications we use are implemented based on Portico, and we observed two issues in the underlying communication mechanism when used with checkpointing and restoring.

Firstly, the communication protocol Portico uses, which is Jgroups UDP multicast, does not work well with CRIU. After restoring the Docker container by the CRIU, the program which uses this communication protocol is not guaranteed to be able to receive messages from its peers. The experiment details to show this problem are shown in APPENDIX A. Thus to address this accidental challenge, we change the communication protocol from Jgroups UDP multicast to Jgroups UDP unicast which is used for communicating between the federates of the co-simulation. The modification code details are shown in APPENDIX B.

Note that in the configuration file, even though the communication protocol is set to be unicast, the Jgroups programs in the same cluster still use a unique multicast IP address for discovering peers. Another adjustment which we had to do in our experiments to enable reliable checkpoint/restore of the docker containers was to use the host machine's network interface for containers to communicate, thus allowing the Jgroups process running inside a container to discover its peers through the UDP multicast protocol. Thus, when there are multiple co-simulations deployed on a cluster of servers, we have to assign each co-simulation a unique multicast IP address.

The second challenge which we encountered to support checkpoint/restore was the heartbeat mechanism that the Portico HLA uses for discovery and fault detection in the co-simulation. Portico programs use the host machine's system clock to calculate the time interval between the arrival of two heartbeat messages received from its peers.

However, the Portico program which gets restored does not take into account the new system time and the fact that it is restored from a snapshot when deciding if it has lost any heartbeat alerts from its peers. For example, after checkpointing and restoring, a Portico program may immediately consider its peers as crashed because the restoring time can be larger than the heartbeat timeout. In Fig 4.1, two Portico programs, A and B, are sending heartbeat messages to each other with a period of 5 seconds. At 10th second, both of them are checkpointed. At 20th second, the checkpointing finishes for both and they continue to run. At 30th second, B crashes and we immediately restore both of them. At 50th second, the restoring finishes for both. In the checkpoint snapshot of A, the last time it received a heartbeat message from B was at 5th second. Thus, after restoring, it considers not receiving the heartbeat message from B for 45 seconds. If this value is larger than the heartbeat timeout, A will make a decision that B has been crashed.

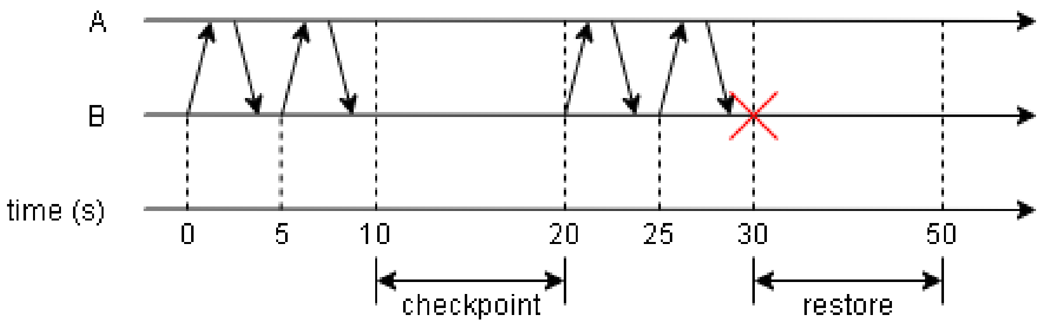

Figure 4.1: Checkpoint and restore 2 Portico programs with the heartbeat mechanism.

To this issue, a simple solution is to remove the heartbeat mechanism. In the Portico HLA, the heartbeat mechanism is used for detecting and handling application-level failures, however, we make an assumption in our study, that there are only Docker container-level failures and no other type of failures. Hence, for this study, we disable the heartbeat messaging alerts. The experiment to show this problem and the technical details of the solution are shown in APPENDIX C.

4.2 Coordinated Checkpoint

In this section, we first present how to checkpoint and restore a single docker container running a co-simulation federate. Next, we show the design and implementation of checkpoint/restore coordination mechanism for distributed containerized co-simulations.

4.2.1 Checkpoint & Restore A Single Container

The Python Script to checkpoint and restore a single container is shown in APPENDIX D. To checkpoint a container, we use the ”*docker checkpoint*” command to save the container's memory. Additionally, we need a way to save the file system of the container. Thus, we use the ”*docker commit*” command to commit it into a new image. To restore the container, we use the ”*docker create*” command to create a new container based on the saved image, then use the ”$docker\ start\ --checkpoint$” command to restore the saved memory on the newly created container. We need to ensure the file system is also restored to the previous checkpoint's state, hence we also checkpoint the file system supporting the docker container.

The ”$docker\ start\ --checkpoint\ --checkpoint-dir$” command can use an external checkpoint directory to restore a container. However, this command in our experimentation was found unreliable for some

Docker versions. Hence, a workaround is to move an external checkpoint directory to the internal container checkpoint directory, and then use the "*docker start −−checkpoint*" command to restore the container.

We observed that there can be only one container being restored at the same time on a given host machine. In our experiments, we observed that if we try to restore multiple containers simultaneously, only one of them can be restored successfully while others would fail. This issue may be due to the checkpointing support as of Docker version 19.03.09, the checkpoint feature is still in the experimental phase. But we believe this issue will be resolved in the upcoming versions of the Docker software.

4.2.2 Coordination Mechanism

Since the co-simulation execution in our experiments follows the BSP computation model, a suitable checkpoint time is when all the federates finish their work in the current time step and are ready to advance to the next time step. Here, we give 2 reasons for this choice of time frame:

1) Since there is no messaging at the federate level during this time, inconsistency problems after restoring co-simulations will be minimized.

2) Since there is no computation at the federate level during this time, the number of memory states needed to store is smaller than any other time point through the co-simulation execution. Thus, the time to checkpoint and restore is shorter and the checkpoint file size is smaller.

The next problem is how to checkpoint the whole system at the desired time point. The approach we present involves pausing all the federates at the end of an execution time period, and thereafter, take a checkpoint. The details are described as follows:

1) The global manager sends a *pause* request to the federation manager.

2) Upon receiving the *pause* request, the federation manager further sends a *pause* signal to other federates in the federation.

3) After finishing the computation task for the current time step, each federate gets paused until receiving a *resume* signal from the federation manager.

4) After waiting for the most time-consuming federate to finish its work, we safely checkpoint the entire federation.

4.3 Incorporating the Checkpoint/Restore Support in EXPPO

This section shows the implementation to extend the EXPPO with checkpoint/restore support. The details to checkpoint, rollback, and detect failure for a co-simulation are shown in Fig 4.2, Fig 4.3, and Fig 4.4,

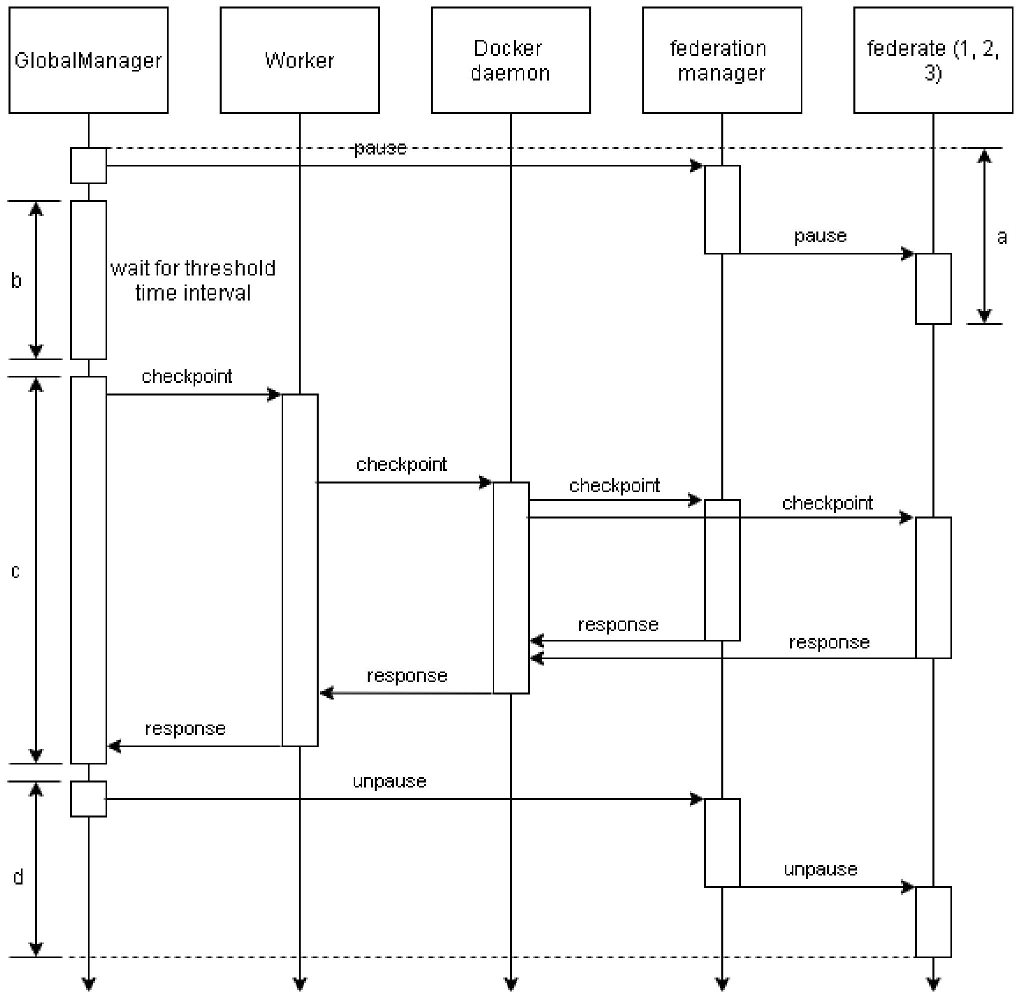

Figure 4.2: Sequence of operations to checkpoint a co-simulation. *GM* is the GlobalManager. *fedmgr* is the federation manager. *fed1*, *fed2* and *fed3* are 3 federates. *fedmgr* and *fed1* are deployed on Worker1 while *fed2* and *fed3* are deployed on Worker2.

respectively. The figures show that the federation manager and three federates are deployed on the same worker node. The description for checkpointing, restoring and failure detection mechanisms are provided next.

1) **Checkpoint**

   a) In Fig 4.2, GlobalManager sends a *pause* request directly to the federation manager. Next, the federation manager sends *pause* interactions to all the federates.

   b) GlobalManager waits some time for the most time-consuming federate to finish the work of the current time step.

   c) GlobalManager sends checkpoint requests to Worker. Then Worker directs the Docker daemon

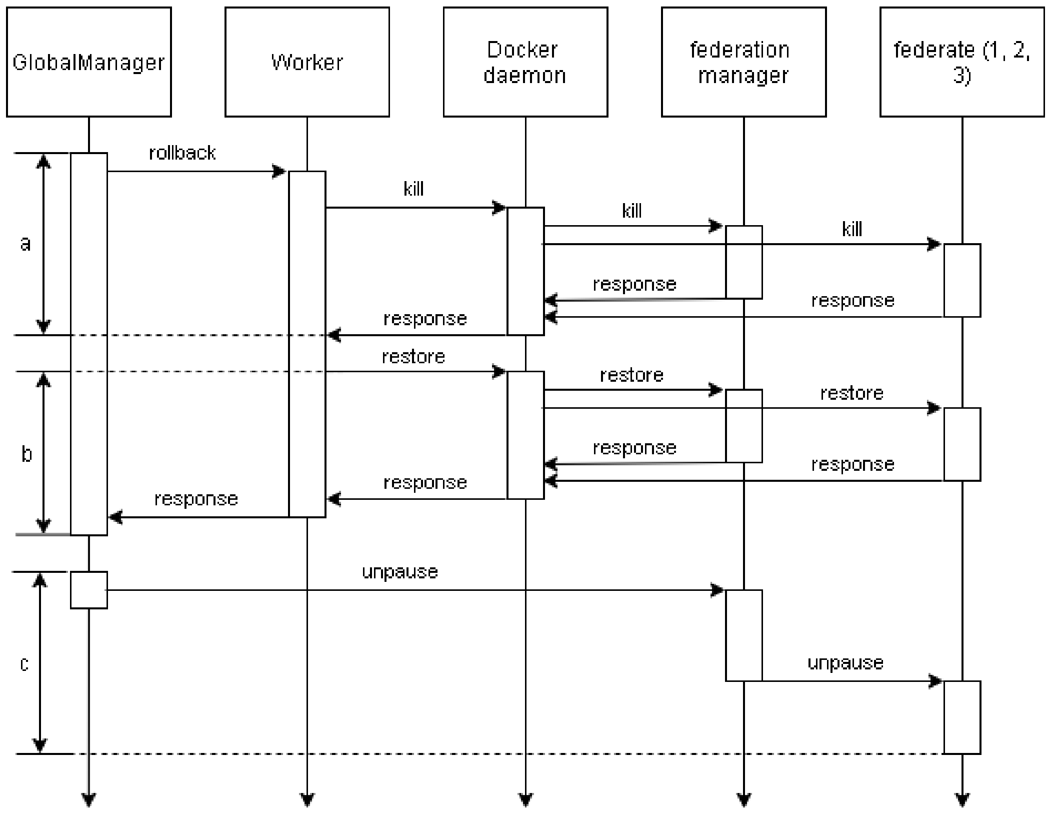

Figure 4.3: Sequence of operations to rollback a co-simulation.

to checkpoint the federation manager and all the federates. After receiving a response from the Docker daemon, Worker also sends a response to GlobalManager

d) GlobalManager sends an unpause request to the federation manager, then the federation manager sends unpause interactions to all the federates.

2) **Rollback**

a) In Fig 4.3, GlobalManager sends a *rollback* request to Worker, then the Worker directs the Docker daemon to kill the federation manager and all the federates. Afterward, the Docker daemon sends a response back to Worker.

b) Worker asks the Docker daemon to restore the federation manager and all the federates. Afterward, the Docker daemon sends a response back to Worker.

c) GlobalManager sends an *unpause* request to the federation manager, then the federation manager sends *unpause* interactions to all the federates.

3) **Failure Detection**

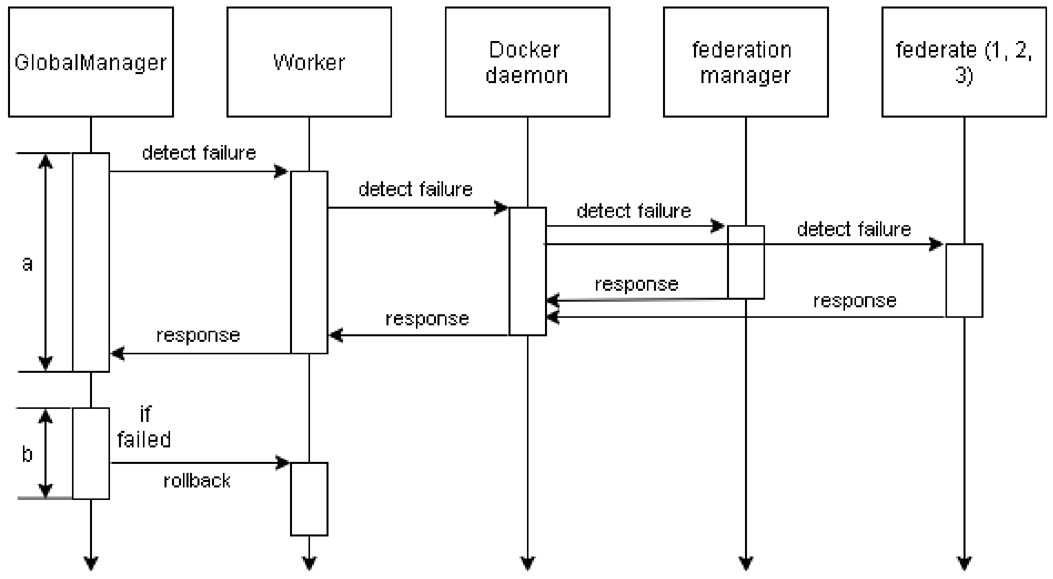

Figure 4.4: Sequence of operations to detect failures in a co-simulation.

a) In Fig 4.4, GlobalManager sends a *failure detection* request to Worker, then the Worker asks the Docker daemon to check the running status of the federation manager and all the federates. Afterward, the Docker daemon sends a response back to Worker and Worker forwards the response to GlobalManager.

b) If there is any container failure, GlobalManager sends a *rollback* request to Worker and restarts the process to rollback the co-simulation.

Algorithm 1 Periodic Checkpointing & Failure Detection

```
1: $federation\_started = false$
2: $current\_time = 0$
3: **while** job not finished **do**
4:     **if** $federation\_started$ **then**
5:
6:         **if** $current\_time\%failure\_detect\_period == 0$ **then**
7:             detect failure, if any, rollback
8:         **else if** $current\_time\%checkpoint\_period == 0$ **then**
9:             detect failure, if any rollback, otherwise checkpoint
10:        **end if**
11:    **else if** send a request to $fedmgr$ for verification and get respond with YES **then**
12:        $federation\_started = true$
13:    **end if**
14:    $current\_time += 1$
15:    sleep for 1 second
16: **end while**
```

After the GlobalManager deploys a co-simulation, it creates a new thread to periodically checkpoint and detects failures of the container federates. The pseudo-code of the newly created thread is shown in Algorithm 1. It first verifies that the co-simulation is running. This is necessary because we want to ensure we checkpoint the co-simulation during it's execution time. After this verification, the thread starts to periodically checkpoint the co-simulation and detects for container failures. Note that each time before we checkpoint the co-simulation, we have to ensure that we are not checkpointing during the moment when failure has occurred in the system. Otherwise, we might checkpoint a failed Docker container. We assume that no failure happens during the time span after the failure detection and before the checkpointing finishes. We also assume that the checkpoint period and failure detection period are both integer values of seconds.

CHAPTER 5

Evaluations

5.1 Experiment Setup

The experimental evaluation is conducted on Chameleon Cloud (Chameleon Cloud, 2021) using 5 homogeneous servers with 4 VCPUs, 8GB RAM, 40G total disks, and Ubuntu 16.04 operating system configuration. The runtime platform is based on the docker engine of version 19.03.9 and containerd version 1.2.6 with swarm mode enabled. The 5 servers are composed of 4 worker nodes and 1 master node on which FE, JM, and GM are deployed.

The simulation job consists of one federation manager and 3 federates each running a unique application from PARSEC benchmark: freqmine, blackscholes, and ferret (et al., 2008). These applications were used in our experiments since they represent real-world simulation tasks. Each application federate executes for 100 logical time steps.

As mentioned in Section 4.1, all the docker containers of each co-simulation use the host machine's network for communication and the GM assigns a unique IP address to each federates of the co-simulation.

5.2 Experiments & Result Analysis

In this section, we evaluate the performance of our approach along following metrics: the additional execution time cost and the time cost to recover from a failure. They are chosen since checkpointing causes additional time cost for co-simulation execution while recovery from a failure takes less time. Comparison of them can help us make a better trade-off.

5.2.1 Checkpoint Different Sizes of Memory

Different co-simulations may have different time costs for checkpointing and restoring. We want to verify that the time cost has a positive correlation with the size of memory needed to save the co-simulation state. In this experiment, we used a Python list with different lengths to represent the program memory with different memory capacities. We used CRIU to checkpoint and restore the program and measure the total time cost. The source code of the Python program we use is shown in APPENDIX E. After creating the list with a certain length, we invoke the following CRIU command to checkpoint it:

```
sudo criu dump -t 23710 -D checkpoint/ -j -v4
```

Here, *dump* is the command to save a process. 23710 is the process ID of the Python program. *checkpoint/*

is the directory to store checkpoint files. *-j* option is used for shell jobs. *-v4* option is used for the highest level of verbosity.

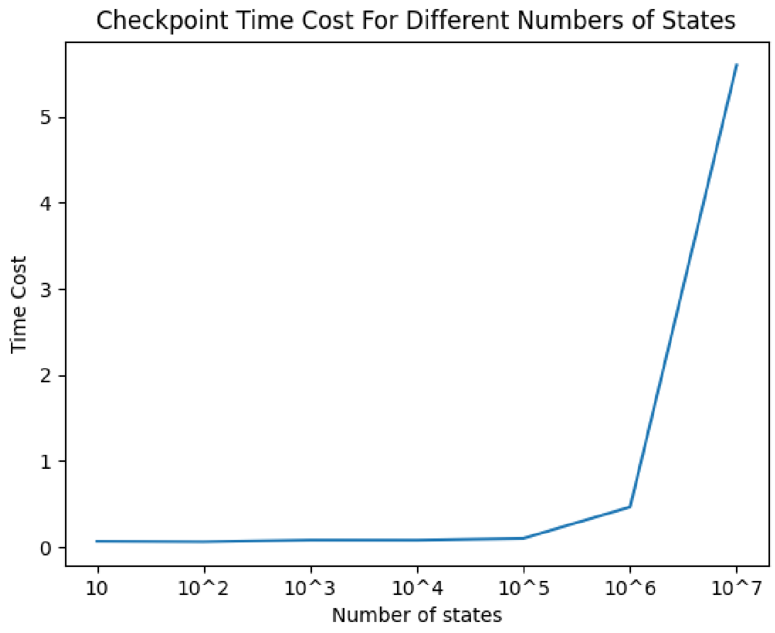

Figure 5.1: Checkpoint time cost for different numbers of memory states.

The time cost measurement for the above command is shown in Fig 5.1. We can see that as the co-simulation memory states increase, the time cost for the checkpointing also increases.

5.2.2 Checkpoint & Restore A Single Federate

The second experiment we did is to measure the time cost for checkpointing and restoring a single application from the PARSEC benchmark in isolation. The average checkpoint & rollback time costs for each application are shown in table 5.1. Each application is checkpointed and restored one by one without any parallel checkpointing so as to avoid issue of resource contention among participating processes while doing checkpointing and restoring of an application.

application	checkpoint	restore
fedmgr	8.384s	7.394s
ferret	5.072s	4.108s
blackscholes	5.258s	4.379s
freqmine	5.783s	4.445s

Table 5.1: Checkpoint & restore time cost for each application from PARSEC benchmark.

To avoid waiting too much time on I/O disk tasks, we can use multi-threading to parallelize the operations to simultaneously checkpoint or restore multiple federates deployed on the same host machine. The best-case scenario to checkpoint or restore a co-simulation is that all the federates and the federation manager are deployed on different servers, and thus, the time cost to checkpoint a co-simulation is the one to checkpoint the most time-consuming single application, which is the federation manager's checkpointing time cost. The worst-case scenario is that all the federates and the federation manager are deployed on the same server, which can have the most resource contention. In this case, as we measured, the average checkpointing and restoring time costs for a co-simulation are 17.749s and 16.841s, respectively. This is also proof that there are indeed resource contentions to checkpoint or restore multiple applications at the same time.

5.2.3 EXPPO's Checkpoint Support

Finally, we measure the additional time cost overhead for EXPPO's checkpoint support and how long it takes to recover in the event of failure.

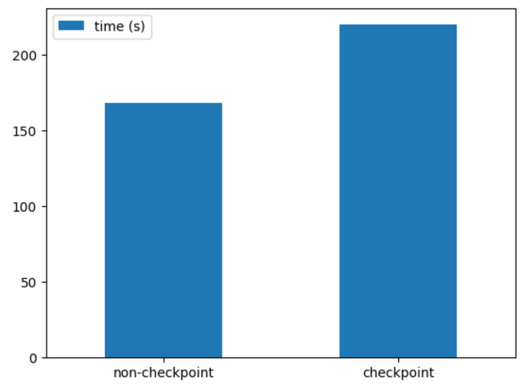

Figure 5.2: Total execution time comparison.

The time intervals for periodic failure detection and checkpoint were set to 10 seconds and 60 seconds, respectively. We injected container failures using the "*docker stop*" command. From our observation, the co-simulation jobs never take more than 3 seconds to finish one logical time step. Thus, the waiting time after sending the pause request when the GM checkpoints a co-simulation is hence set to be 5 seconds. In one co-simulation, the federation manager and all the federates are deployed on the same server for the worst-case measurement.

To measure the additional time cost for periodical checkpointing, we assume there is no failure through the execution. The original average total co-simulation execution time was measured as 168s. The average total execution time with checkpoint support was measured as 220s. Thus, the additional time cost for checkpointing is 220s - 168s = 52s, which means the checkpoint support can cause around 30% overhead for execution. The time cost comparison is shown in Fig 5.2.

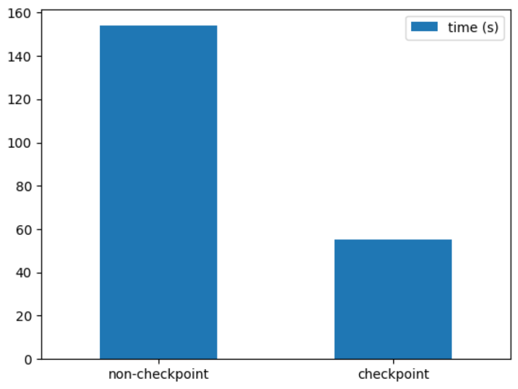

Figure 5.3: Recovery time comparison.

To measure the recovery time, we assume that there is only one failure during the entire execution time of the co-simulation, and the probability for the failure to appear is the same at any time point through the execution. To recover the co-simulation successfully, the GM first detects that the failure has occurred, and then rollback the co-simulation to the last successful checkpoint state. The average total execution time is measured as 275s. Thus, the recovery time can be calculated as 275s - 220s = 55s, where 220s is the total execution time without failures.

If we decide to completely restart the co-simulation once detecting a failure, the co-simulation has to go through the preparation time again before execution, which is around 70s on average. Since the probability of failure happening is the same at any time point, the previously failed execution takes half of the total execution time on average, which is 168s / 2 = 84s. As a result, the recovery time cost is 70s + 84s = 154s on average, which is as 3 times larger as the one with checkpoint support. The recovery time cost comparison is shown in Fig 5.3.

CHAPTER 6

Conclusion & Future Work

To enhance the resilience of distributed containerized co-simulations, we demonstrated the effectiveness of *Checkpoint/Restart* (C/R) technique to provide fault tolerance for running co-simulations in a cloud computing environment. To successful checkpoint and restore a co-simulation, there needs to be a reliable and coordinated communication mechanism to achieve distributed checkpointing. For the underlying communication mechanism, we updated the JGroups from the default UDP multicast protocol into UDP unicast for communication between the federates of the co-simulation. We also disabled the Portico heartbeat mechanism to allow for checkpoint and restore to operate successfully. For the checkpoint coordination mechanism, we designed a co-simulation pause strategy to pause all the federates at the end of a simulation time step for doing checkpointing the federates. Furthermore, we extended EXPPO with checkpoint support so that the co-simulation can be checkpointed periodically and any failure can be automatically detected and appropriate restore action can be executed. Our experiment results demonstrate the effectiveness of our approach. Although in our experimental run, the periodic checkpointing incur a 30% overhead time cost for the co-simulation execution, the recovery time is almost 3 times faster than completely restarting the co-simulation from the start.

Our future work will explore the following concrete problems:

1) In the current pause mechanism for the co-simulations, after sending the pause request, the GM needs to wait for the longest time-consuming application to finish the work of the current time step. However, in the current setup, we assume that we know what is the longest execution time a priori. Thus, there is a need to dynamically identify when the execution of all the federates for a given time-step is complete, before doing the checkpointing.

2) Furthermore, we currently assume that the failure can occur only at the container level. However, there are many other types of failures, such as network failures, and application-level failures which also need to be handled. Thus our future work will be to devise strategies to mitigate such failures.

3) Currently, the checkpoint and failure detection time intervals are set to be 60s and 10s, respectively. However, these time intervals are not optimal. Thus there is a need to study and make appropriate decisions to select these time periods such that they provide better resiliency and run-time for the co-simulations.

Appendix A

Experiment to Show The Jgroups UDP Multicast's Probelm With CRIU

A.1 Experiment Setup

1) **Ubuntu:** 16.04

2) **CRIU:** 3.15

3) **Jgroups:** 4.0.0.Final. The jar file can be downloaded as following:

   https://sourceforge.net/projects/javagroups/files/JGroups/4.4.0.Final/

The source code of the test Java program is shown as follow:

```java
import org.jgroups.JChannel;
import org.jgroups.Message;
import org.jgroups.View;
import org.jgroups.ReceiverAdapter;
import java.io.*;

public class Example extends ReceiverAdapter
{
    JChannel channel;
    String user_name=System.getProperty("user.name", "n/a");

    private void start() throws Exception {
        channel=new JChannel().setReceiver(this);
        channel.connect("ExampleFederation");
        eventLoop();
        channel.close();
    }

    private void eventLoop() {
        BufferedReader in=new BufferedReader(new InputStreamReader(System.in));

        while(true) {
            try {
                System.out.print("> "); System.out.flush();
                String line=in.readLine().toLowerCase();
                if(line.startsWith("quit") || line.startsWith("exit"))
                    break;
                line="[" + user_name + "] " + line;
                Message msg=new Message(null, line);
                channel.send(msg);

                Thread.sleep (1000);
            }
            catch(Exception e) {
            }
        }
    }

    @Override
    public void viewAccepted(View new_view) {
        System.out.println("** view: " + new_view);
    }

    @Override
    public void receive(Message msg) {
        System.out.println(msg.getSrc() + ": " + msg.getObject());
    }

    public static void main(String[] args) throws Exception {
        new SimpleChat().start();
    }
}
```

A.2 Experiment steps & Results

1) On a Ubuntu 16.04 operating system, prepare 3 terminals, A, B and C.

2) In terminal A and B, use the following command to run 2 Java programs with Jgroups UDP multicast. *jgroups-4.0.0.Final.jar* is the Jgroups jar file. *Example.java* is the test java program.

```
java -Djava.net.preferIPv4Stack=true -cp jgroups-4.0.0.Final.jar
Example.java
```

3) Type anything in either terminal A or B. Then we can see the same thing appears in the other terminal.

4) In terminal C, use the following command to find out the process IDs of 2 programs. In our case, they are 3770 and 3781.

```
ps -ef | grep \java -cp jgroups-4.0.0.Final.jar Example.java
```

5) In terminal C, use the following command to create 2 new directories. They will be used later to store the checkpoint files.

```
mkdir checkpoint1 checkpoint2
```

6) In terminal C, use the following command to checkpoint 2 programs using CRIU.

```
sudo criu dump -t 3770 -D checkpoint1/ -j -v4 --tcp-established &&
sudo criu dump -t 3781 -D checkpoint2/ -j -v4 --tcp-established
```

7) In terminal A and B, use the following 2 commands to restore both programs.

```
sudo criu restore -D checkpoint1/ -j
sudo criu restore -D checkpoint2/ -j
```

7) In terminal A or B, type something again. At this time, the other one will not show anything. This means there are some problems with their communication.

Appendix B

Change Jgroups UDP Multicast to UDP Unicast

B.1 The Configuration File

Portico uses the *jgroups-udp.xml* file to configure Jgroups. The original file can be found here:

https://github.com/openlvc/portico/blob/master/codebase/resources/jars/portico.jar/etc/jgroups-udp.xml

It is shown as follow:

```
<config xmlns="urn:org:jgroups"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:org:jgroups http://www.jgroups.org/schema/JGroups-3.1.xsd"
    >

    <UDP bind_addr="${portico.jgroups.udp.bindAddress:SITE_LOCAL}"
         mcast_group_addr="${portico.jgroups.udp.address:239.20.9.13}"
         mcast_port="${portico.jgroups.udp.port:20913}"
         mcast_recv_buf_size="${portico.jgroups.udp.receiveBuffer:25M}"
         mcast_send_buf_size="${portico.jgroups.udp.sendBuffer:640K}"
         ucast_recv_buf_size="8M"
         ucast_send_buf_size="640K"

         ip_mcast="true"
         enable_diagnostics="false"

         bundler_type="transfer-queue"
         max_bundle_size="${portico.jgroups.bundling.maxSize:64K}"

         thread_pool.enabled="true"
         thread_pool.min_threads="2"
         thread_pool.max_threads="8"
         thread_pool.keep_alive_time="5000"
    />

    <!-- Discovery protocol. Multicast requests for other memgers on the local network. --
    >
    <PING break_on_coord_rsp="true"/>

    <!-- Detect when a federate has gone tits-up with a simple heartbeat algorithm.
         We can turn on countMessages which will treat all messages received from a
         federate as a heartbeat, but this add overhead and apparently isn’t
         recommended. Leave it off for now -->
    <FD_ALL/>

    <!-- When we think we have a dead federate, as determined by the FD protocols,
         we need to finally verify this. VERIFY_SUSPECT will ping the suspect,
         waiting at most timeout millis before giving up and calling it dead -->
    <VERIFY_SUSPECT timeout="500"/>

    <!-- NAKACK2 provides the reliable, ordered properties to our comms -->
    <pbcast.NAKACK2 xmit_interval="500"
                    xmit_table_num_rows="100"
                    xmit_table_msgs_per_row="2000"
                    xmit_table_max_compaction_time="30000"
                    use_mcast_xmit="false"
                    discard_delivered_msgs="true"/>

    <!-- UNICAST3 delivers reliable and ordered messaging for unicast UDP messages -->
    <UNICAST3 xmit_interval="500"
              xmit_table_num_rows="100"
              xmit_table_msgs_per_row="2000"
              xmit_table_max_compaction_time="60000"
              conn_expiry_timeout="0"/>

    <!-- RSVP causes message send with the RSVP header to block the sending call
         until all recipients have acknowledged they got it. If ack_on_delivery
         is set true, an ACK is only be sent after the message has been processed -->
    <RSVP ack_on_delivery="true" throw_exception_on_timeout="true"/>

    <!-- The STABLE protocol helps ensure only the minimum number of messages are
         kept around in case they are needed for retransmission.  -->
    <pbcast.STABLE max_bytes="4M"
                   desired_avg_gossip="60000"/>

    <!-- GMS provides generation Group Memebership Services, handling group join
         requests, view updates and changes and anything membership related -->
    <pbcast.GMS print_local_addr="true"
                join_timeout="${portico.jgroups.gms.jointimeout:3000}"/>

    <!-- Flow control prevents any single sender overwhelming slower receivers
         with messages which could cause them to be dropped -->
    <UFC max_credits="2M" min_threshold="0.4"/>
    <MFC max_credits="${portico.jgroups.flow.credits:2M}"
         min_threshold="${portico.jgroups.flow.threshold:0.4}"/>

    <!-- FRAG2 breaks up large messages into smaller sizes so they can be send in chunks -->
    <FRAG2 frag_size="${portico.jgroups.frag.size:60K}"/>

    <!-- FLUSH forces all group members to send all pending messages prior to
         some event such as the joining of a new member -->
    <pbcast.FLUSH retry_timeout="3000"/>
</config>
```

B.2 Modification Details

The changes we did are shown as follow:

1) In line 13, set the variable *ip_mcast* to be false.

2) Remove line 6-9.

3) We changed the line 26 into the following 2 lines:

```
    <MPING mcast_addr="${portico.jgroups.udp.address:239.20.9.13}"
        Mcast_port="${portico.jgroups.udp.port:20913}"/>

```

Appendix C

Experiment to Show Portico Heartbeat Mechanism's issues with Checkpoint and restore and the proposed Solution to tackle this issue.

C.1 Experiment Setup

CRIU and Ubuntu versions are the same as Appendix A. We can get Portico from the following commands.

1) Use the following command to get Portico.

```
git clone https://github.com/openlvc/portico
```

2) Change the *jgroups-udp.xml* file as instructions in APPENDIX B.

3) Use the following command in the directory *codebase/* to rebuild the project.

```
./ant sandbox
```

C.2 Experiment steps & Results

1) On a Ubuntu 16.04 operating system, we prepare 3 terminals, A, B and C.

2) In A and B, in directory *codebase/dist/portico-2.1.0/examples/java/hla13/*, use the following 2 commands respectively to run the sample federate Portico provides.

```
sh linux.sh execute fed1
sh linux.sh execute fed2
```

3) Wait some time until both programs ask for user input to continue.

4) In terminal C, run the following command to get the process IDs of 2 programs. In our case, they are 9654 and 9687.

```
ps -ef | grep java
```

5) In terminal C, use the following command to create directories to store checkpoint files later.

```
mkdir checkpoint1 checkpoint2
```

6) In terminal C, use the following 2 commands to checkpoint both programs.

```
sudo criu dump -t 9654 -D checkpoint1/ -j -v4 --tcp-established
sudo criu dump -t 9687 -D checkpoint2/ -j -v4 --tcp-established
```

7) Wait for 1 minute. Then in terminal C, use the following command to restore one program.

```
sudo criu restore -D checkpoint2/ -j
```

8) In terminal C, we can see the program immediately complains that the other one has crashed.

C.3 Solution

We set the heartbeat timeout to be a set at a higher threshold level,beyond which we do not expect to restore the checkpoint state. In line 37, we changed the number from 500 to 5,000,000,000.

Appendix D

Checkpoint Python Script

```
def checkpoint (client, container_obj, container_name, checkpoint_name, image_name):
    # Remove the last checkpoint first
    try:
        os.system ('docker checkpoint rm --checkpoint-dir /tmp/docker_checkpoint ' + container_name + ' ' + checkpoint_name)
    except Exception:
        print ('Cannot remove last checkpoint for %s'%container_name)
    os.system ('docker checkpoint create --leave-running --checkpoint-dir /tmp/docker_checkpoint ' + container_name + ' ' + checkpoint_name)
container_obj.commit (repository = image_name)

def delete_container (container_obj):
    container_obj.remove (force=True)

def create_container(client, image, name, network=None, ip_address=None, command=None, cpuset_cpus=None, mem_limit=None, volumes=None, environment=None, tty=True):
    client.containers.create (
        image=image,
        name=name,
        network=network,
        cpuset_cpus=cpuset_cpus,
        mem_limit=mem_limit,
        volumes=volumes,
        command=command,
        environment=environment,
        tty=tty
    )

    return client.containers.get (name)

def move_checkpoint_file (cid, checkpoint_name):
    subprocess.call (['mv', '/tmp/docker_checkpoint/' + checkpoint_name, '/var/lib/docker/containers/' + cid + '/checkpoints'])

def start_from_checkpoint (pre_cid, name, checkpoint_name):
    subprocess.call (['docker', 'start', '--checkpoint', checkpoint_name, name])
```

```

lock = threading.Lock ()
def rollback (client, container_obj, cid, name, image, checkpoint_name, command=None, tty=
    True, environment=None, volumes=None, cpuset_cpus=None, mem_limit=None, network=None,
    ip_address=None):
    delete_container (container_obj)
    container_obj = create_container (client=client, name=name, image=image, command=
    command, tty=tty, environment=environment, volumes=volumes, cpuset_cpus=cpuset_cpus,
    mem_limit=mem_limit, network=network, ip_address=ip_address)
    move_checkpoint_file (container_obj.id, checkpoint_name)
    # Without lock, unknown problems will happen
    with lock:
        start_from_checkpoint (container_obj.id, name, checkpoint_name)

def check_alive (containers):
    for container in data:
        container_obj = worker.__docker_client.containers.get (container)
        if container_obj.attrs['State']['Status'] != 'running':
            return 'failed', 200

    return 'running', 200
```

Appendix E

The Script to Create Different Sizes of Python Lists

```python
import sys
import time

states = int (sys.argv[1])
a = []
for i in range (states):
  a.append (i)

print ('Finished generating %d states'%states)
time.sleep (200)
```

References

Berg, G. and Brattl, M. (2017). Checkpointing with docker containers in high performance computing.

Chameleon Cloud (2021). https://chameleoncloud.org/.

CRIU (2021). https://www.criu.org/main.

Docker (2021). https://www.docker.com/.

et al., A. (2020). Docker container deployment in distributed fog infrastructures with checkpoint/restart. *2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*.

et al., C. B. (2008). Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. *2008 IEEE International Symposium on Workload Characterization*.

et al., P. (2019). Checkpointing and migration of iot edge functions. *19: Proceedings of the 2nd International Workshop on Edge Systems, Analytics and NetworkingMarch, 2019, Pages 60–65*.

et al., T. (2000). The heterogeneous bulk synchronous parallel model. *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*.

Group, H. E. W. (2010). Ieee standard for modeling and simulation (m&s) high level architecture (hla) framework and rules.

Guermouche, A., Ropars, T., Brunet, E., Snir, M., and Cappello, F. (2011). Uncoordinated checkpointing without domino effect for send-deterministic mpi applications. *2011 IEEE International Parallel & Distributed Processing Symposium*.

H.Zhang, N.Chen, Y.Tang, and B.Liang (2019). Multi-level container checkpoint performance optimization strategy in sddc. *Proceedings of the 2019 4th International Conference on Big Data and Computing*.

Jgroups (2021). http://www.jgroups.org/.

Neves, N. and Fuchs, W. (1998). Coordinated checkpointing without direct coordination. *Proceedings. IEEE International Computer Performance and Dependability Symposium. IPDS'98 (Cat. No.98TB100248)*.

Portico (2021). https://github.com/openlvc/portico.

Schulz, M., Bronevetsky, G., Fernandes, R., Marques, D., Pingali, K., and Stodghill, P. (2004). Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*.

Tamir, Y. and Séquin, C. H. (1984). Error recovery in multicomputers using global checkpoints. *1984 International Conference on Parallel Processing*.

Y.D.Barve, Neema, H., Z.Kang, H.Sun, A., and T.Roth (2020). Exppo: Execution performance profiliing and optimization for cps co-simulation-as-a-service. *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*.