ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

DCfRSM: Fault-Tolerant Checkpoint Approach for Replicated State Machines

M.Sc. THESIS

Niyazi Ozdinc CELIKEL

Department of Computer Engineering

Computer Engineering Programme

JUNE 2020

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE
ENGINEERING AND TECHNOLOGY

DCfRSM: Fault-Tolerant Checkpoint Approach for Replicated State Machines

M.Sc. THESIS

Niyazi Ozdinc CELIKEL
(504171522)

Department of Computer Engineering

Computer Engineering Programme

Thesis Advisor: Assoc. Prof. Dr. Tolga Ovatman

JUNE 2020

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

DCfRSM: Eşlenmiş Durum Makinalarında Hata Kabul Edebilirliği Yüksek Kontrol Noktası Tutma Yaklaşımı

YÜKSEK LİSANS TEZİ

Niyazi Ozdinc CELIKEL
(504171522)

Bilgisayar Mühendisliği Ana Bilim Dalı

Bilgisayar Mühendisliği Programı

Tez Danışmanı: Doç. Dr. Tolga OVATMAN

HAZIRAN 2020

Niyazi Ozdinc CELIKEL, a M.Sc. student of ITU Graduate School of Science Engineering and Technology 504171522 successfully defended the thesis entitled "DCfRSM: Fault-Tolerant Checkpoint Approach for Replicated State Machines", which he/she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Assoc. Prof. Dr. Tolga Ovatman** Istanbul Technical University	....
Jury Members :	**Assoc.Prof.Dr.Şerif Bahtiyar** Istanbul Technical University	....
	Asst.Prof.Dr.Tolga Ensari Istanbul University Cerrahpasa	....

Date of Submission : **15 June 2020**
Date of Defense : **13 July 2020**

To my family,

FOREWORD

I would like to express my appreciation to my supervisor Asst. Prof. Dr. Tolga Ovatman. He always convinced me to do my best during this thesis. Whenever I stuck during research and development activities, I could always get magnificent help from him. Moreover, I could publish an international research paper with his guidance. I will be always proud of being a student of him.

I would also thank my company Siemens Turkey for sponsoring me during publishing my research paper. I would also thank to the scientific and technological research council of Turkey (TUBITAK) for their great support within the project numbered 118E887.

I dedicate this thesis to my dear family. I am grateful for everything they have done for me up until now.

June 2020

Niyazi Ozdinc CELIKEL
Computer Engineer

TABLE OF CONTENTS

Page

FOREWORD ix
TABLE OF CONTENTS xi
ABBREVIATIONS xiii
SYMBOLS xv
LIST OF TABLES xvii
LIST OF FIGURES xix
SUMMARY xxi
ÖZET xxiii
1. INTRODUCTION 1
2. RELATED WORK 5
3. DISTRIBUTED CHECKPOINTING 9
3.1 Background Information 9
3.2 DCfRSM as a Checkpointing Solution 10
4. SYSTEM ARCHITECTURE AND EXPERIMENTAL ENVIRONMENT 15
4.1 Overall Architecture 15
4.1.1 Development environment and development artifacts 15
4.1.2 Context of state machines 15
4.1.2.1 States and state transitions 15
4.1.2.2 Actions 17
4.1.2.3 Listeners 17
4.1.2.4 Factories 18
4.1.2.5 Annotations 18
4.1.3 Execution of state machines 18
4.1.4 Checkpoint artifacts 20
4.2 Interprocess Communication 21
4.3 Implemented Solutions 22
4.3.1 Proof of concept study 22
4.3.2 Conventional solution 22
4.3.3 Centralized solution 23
4.3.4 DCfRSM 24
4.3.5 Mirrored solution 25
4.4 Simulation Environment 25
4.4.1 Infrastructure 25
4.4.2 Orchestration and automation 27
5. EXPERIMENTS AND EVALUATION 31
5.1 Test Bed 31
5.2 Results 33

5.3 Evaluation 36
6. CONCLUSIONS AND FUTURE WORK 41
REFERENCES 43
APPENDICES 45
APPENDIX A 47
APPENDIX B 55
APPENDIX C 59
CURRICULUM VITAE 63

ABBREVIATIONS

DCfRSM	: Distributed Checkpointing for RSM
IaaS	: Infrastructure as a Service
IDE	: Integrated Development Environment
JDK	: Java Development Kit
JVM	: Java Virtual Machine
RAID	: Redundant Array of Independent Disks
REST	: Representational State Transfer
RSM	: Replicated State Machine
UML	: Unified Modeling Language
ZK	: Apache Zookeeper

SYMBOLS

GB : Gigabyte

MB : Megabyte

LIST OF TABLES

Page

Table 5.1 : Experiments.. 32
Table 5.2 : Average seconds of restore durations regarding 4,6,8 and 10 replicas 33
Table 5.3 : Average KB of memory footprint regarding 4,6,8 and 10 replicas..... 37

LIST OF FIGURES

Page

Figure 4.1 : State transitions.......... 16
Figure 4.2 : Overall architecture. 17
Figure 4.3 : Docker instructions.......... 26
Figure 4.4 : Healtcheck. 27
Figure 4.5 : docker-compose.yaml and network instructions. 28
Figure 4.6 : Network instructions.......... 29
Figure 5.1 : ObjectSizeFetcher library. 31
Figure 5.2 : RSM instances in cloud for the test bed with 4 replicas.......... 33
Figure 5.3 : Memory footprint consumption.......... 34
Figure 5.4 : Restore duration.......... 35
Figure 5.5 : Phases for restore duration. 36
Figure 5.6 : Memory footprint from the experiments in cloud. 37
Figure 5.7 : Restore duration from the experiments in cloud. 39
Figure B.1 : UML diagram for the conventional solution. 55
Figure B.2 : UML diagram for the DCfRSM. 56
Figure B.3 : UML diagram for the mirrored solution.......... 57

DCfRSM: FAULT-TOLERANT CHECKPOINT APPROACH FOR REPLICATED STATE MACHINES

SUMMARY

As a result of attractive offers including serverless hosting services by cloud vendors, usage of state machine models has been widely spread in various areas in terms of software development for cloud. Serverless computing phenomenon offers software developers not only eliminating the need to take care of many aspects of software development stack but also focusing on developing main tasks to be executed by cloud system. Using state machine models in order to express the required functionality in one of the popular approaches for serverless computing.

Cloud vendors also offers non functional requirements such as fault tolerance, integrability, etc. Replicating a single state machine across cluster which is going to work in distributed manner with the aim of handling requests is very well-known approach for providing fault tolerance. With other words, a major approach that can be used in fault tolerance is checkpointing. Checkpointing offers saving the system state in specific frequencies in order to boot the system back in case of any failure. Especially, applying different checkpointing approaches in replicated state machine systems is an highly active research topic for today's cloud computing era.

In this thesis, advantages of splitting snapshots of replicated state machine images with the aim of decreasing memory overhead during persisting the system context is investigated. This type of checkpointing offers deploying and serializing of whole request history into many parts during persisting checkpoints, and hence, each of the replicated state machine instances are going to store a specific piece of history instead of full execution history while overall history is going to be gathered from the overall system, which also means, logical master history will be shared between all the active state machine replicas inside current cluster.

Within this thesis, *Java Spring State Machine* framework is used to model checkpointing mechanism and benchmarked results with many other approaches such as *Centralized Approach*, *Conventional Approach* and *Mirrored Approach*. By doing so, distributed snaphots can be gathered, merged and serialized during state machine execution even if more than one replica fails at the same time. It proved that by using proposed approach, the amount of memory overhead needed for persisting checkpoints is decreased. In contrast to this advantage, recovery time which expresses the time interval needed for a new replica to be joined into cluster is increased due to extra communication need for collecting partial histories from replica instances.

DCfRSM: EŞLENMİŞ DURUM MAKİNALARINDA HATA KABUL EDEBİLİRLİĞİ YÜKSEK KONTROL NOKTASI TUTMA YAKLAŞIMI

ÖZET

Bulut bilişim alanında faaliyet gösteren markaların sunduğu sunucusuz çözümler mimarisi dahilinde, yazılım geliştirme süreçlerinde durum makinalarının kullanım sıklığı artmıştır. Sunucusuz çözümler mimarisi, sadece yazılım geliştirme süreçlerindeki birçok alana hakim olma gerekliliğini ortadan kaldırmamış, aynı zamanda yazılımcılara ana geliştirme konularına odaklanma fırsatı vermiştir. Geliştirilen servislerin yüksek erişilebilir olmasını sağlamak, testlerin otomatize bir şekilde koşmasını sağlamak, daha az efor ile canlı sistemi güncellemek, yük testleri koşturmak, canlı sistemlerden test sistemlerin beslenmesini otomatik hale getirmek, canlı sistemlerde yaşanan sorunları anında gözleyebilmek ve sorunlara anında ve otomatik olarak müdahale edilmesini sağlamak, canlı sistemlerde kulllanıcıları yetkilendirmek gibi konuların belirtilen servisler tarafından sağlanabiliyor oluşu, yazılımcalara ürüne ve ürün geliştirme konularına daha fazla hakim olmasını sağlayacak bir fırsat sunmaktadır. Yukarıda belirtilen özelliklerin sağlanması amacıyla kullanılan popüler yaklaşımlardan biri, durum makinalarının kullanımıdır.

Bulut bilişim alanında faaliyet gösteren markalar, çözümleri içerisine birçok işlevsel olmaktan uzak özellik eklemiştir, bunlara örnek olarak hata kabul edilebilirliği, bütünleşebilirlik, yüksek erişilebilirlik verilebilir. Bütünleşebilirlik, servislerin başka platforma aktarılmasındaki kolaylığın ölçüsüdür. Platformlar arası kolayca geçişi yapılabilen servislerin, bütünleşebilirlik özelliği yüksek demektir. Yüksek erişilebilirlik ise, bir uygulama veya servisin her zaman ayakta olması ve iyi/kötü bir cevap verebiliyor olmasıdır. Burada dikkat çekilmesi gereken nokta, servisin ayakta oluşunun yüksek erişilebilirlik kriterinin sağlandığı anlamına gelmediğidir. Servis ayakta olabilir, gelen çağrıları kabul edilebilir fakat çağrıyı yapan programa/kullanıcıya iyi veya kötü bir cevap dönebilyor olması gerekmektedir. Hata kabul edilebilirlik ise, bir servisi oluşturan parçaların herhangi birinde yazılımsal veya donanımsal bir hata oluştuğunda, bunun servisi kullanan taraflara farkettirilmeden algılanması ve problemin çözülmesi gerekliliğini kapsar. Servisi oluşturan parçalardan biri cevap veremez duruma geçtiyse, bu sorunlu parçanın sistem bağlantılarını koparıp yerine aynı özellikte ve aynı işlevde bir parçanın konulması gerekliliği, yüksek erişilebilirliğin kriterlerinden birisi olarak karşımıza çıkmaktadır. Bu faktörlerin hepsi gözönüne alındığında, sistemin kendisinin ve tek tek tüm bileşenlerinin *yedekliliği* gerekliliğinin önemi anlaşılmaktadır. Sistemin ve parçalarının yedeklerinin tutulması kadar, bu yedeklerin işlevliliği, yani bu yedekten geri dönebilme başarısı da dikkate alınmalıdır. Hatta bu ihtiyaçtan ötürü, büyük firmalar, sene içinde alınan sistem yedeklerinin işlevselliğini görmek adına, *yedekten dönüş* testleri yaparak, belirli bir andaki sistem yedeğine dönerek sistemlerin erişilebilirliğini ve işlevselliğini test etmektedir. *Kontrol noktası* tutma yaklaşımı da, bir yedekleme yöntemi olarak görülebilmektedir.

Bir yazılım servisinin parçalarını oluşturan bilgisayar kümesi içindeki bir makinayı, küme içerisinde dağıtılmış bir şekilde kopyasını çıkarmak adına yapılan çalışmalar, hata kabul edilebilirliğini sağlamak adına yürütülmektedir. Başka bir şekilde söylemek gerekirse, bu amaçla kontrol noktası saklamak, hata kabul edilebilirliğini sağlamak adına uygulanabilir. Kontrol noktasının hangi sıklıkla saklanacağı kadar, bu noktanın hangi depolama bölümünde -disk, bellek,kartuş,vs- saklanacağı, sistemin geri dönüş süresini etkilemesi bakımından önem arz etmektedir. Kontrol noktası saklama yolu ile sistem durumunu belirli aralıklarla saklamak ve hata durumunda tüm sistemi geri getirmek mümkün olabilmektedir. Yaklaşımlardan biri, tüm sistemin bir bütün halinde kontrol noktasını saklamaktır. Bu yolla, herhangi bir hata durumunda tek bir noktadan geri getirme işlemleri başlatılabilir. Bu yaklaşımın dezavantajı, tutulan kontrol noktası büyük olacağı için geri dönüş zamanının fazla olacağıdır. Diğer yaklaşım ise, tüm sistemin bütün parçalarının ayrı ayrı kontrol noktalarını tutmaktır. Bu yaklaşım daha granüler bir yaklaşımdır ve daha hızlı geri dönüş olanağı verebilir. Günümüzde, bulut bilişim alanında yürütülen araştırmalar dahilinde, kontrol noktası tutma yaklaşımlarının eşlenmiş durum makinaları ile kullanımı, üzerinde çok durulan bir konudur.

Eşlenmiş durum makinaları, servise gelen tüm isteklerin serileştirildiğinin ve sistemden cevap üretildiği an, sisteminin durumunun kayıt edildiğinin garantilendiği sistemlerdir. Dolayısıyla, sistem bileşenlerinin birbirinden haberdar olması, aynı isteğe farklı zamanlarda farklı cevaplar üretilmediğinin garanti altına alınması, yukarıda da anlatıldığı üzere, hata toleransının sağlanması gereken sistemler olduğu açıktır. Bunu sağlamak adına, sistemin tüm bileşenleri, birbirinin *eşleniği* gibi davranmak zorundadır. Bunu sağlamanın en kolay yollarından biri, bir durum makinasının farklı sunucular üzerinde eşlenik bir şekilde çalıştırılmasıdır. Simüle edilmek istenirse, sistemdeki tüm bilgisayarların belli etkenlere/uyarıcılara aynı tepkiyi vererek, aynı anda durumlarını değiştirerek aynı çıktıyı üretmelerinin sağlandığı sistemler olarak düşünülebilir. Eşlenmiş durum makinaları için de farklı kontrol tutma yaklaşımları söz konusudur, bu alanda geniş çalışmalar mevcuttur. Örnek yaklaşımlardan biri, *artan kontrol noktası tutma* yaklaşımıdır. Bu çözümde, sistemin belirli bir anda, tüm bağlamı, kontrol noktası olarak saklandıktan sonra bu noktaya bir işaret konulur, ardından her bir durum geçişinde işaret konulan nokta ile mevcut nokta arasındaki farkı kayıt edilir.

Bu tez çalışması dahilinde, eşlenmiş durum makinalarının sistem bağlamlarını kayıt ederken yaşanan bellek kullanımlarının azaltılması amacıyla kontrol noktalarının dağıtılması yaklaşımının avantajları gösterilmiştir. Önerilen yaklaşım kullanıldığı takdirde, bilgisayar kümesi içerine gelen tüm isteklerin tarihçesinin belli parçalara bölünmesi sağlanacaktır. Dolayısıyla, her bir bir eşlenmiş durum makinası, sisteme gelen isteklerin belirli bir kısmına kontrol noktası olarak kendi sistem bağlamında saklayacaktır. Bunun anlamı ise, sisteme gelen tüm isteklerin tarihçesinin, bilgisayar kümesi içerisinde aktif olarak çalışan durum makinalarının hepsi tarafından paylaşılmasıdır.

Tez çalışması içerisinde, *Java Spring State Machine* kütüphanesi kullanılmış, farklı kontrol noktası tutma yaklaşımları gerçeklenerek sonuçları karşılaştırılmıştır, ki bu yaklaşımlar *Merkezi Yaklaşım*, *Geleneksel Yaklaşım* ve *Yansımalı Yaklaşım* olarak belirlenmiştir. Önerilen çözüm kullanıldığı takdirde, mevcut bilgisayar kümesi içerisinde birden fazla makina bozulsa dahi sistemde ayrık olarak tutulan

kontrol noktaları birleştirilip serileştirilme imkanı doğmaktadır. Çalışma dahilinde gösterilmiştir ki, önerilen yaklaşım kullanıldığı takdirde sistem durumunu saklarken gereken bellek tüketim miktarı azaltılabilmiştir. Bu avantaja karşıt olarak, yeni bir durum makinası bilgisayar sistemine katılırken gereken zamanda bir artış söz konusudur, çünkü bu durum makinası, sistemde aktif olan durum makinalarına dağıtılmış kontrol noktalarını okurken bir zaman kaybı gözlemlenmiştir.

1. INTRODUCTION

In order not to place a high burden on the developer of the cloud service, in today's cloud computing era, there are numerous non functional requirements needed to be satisfied by vendors, such as automatic scaling, fault tolerance, consistency, speed etc. As [1] stated, in order to increase the quality of user experience, RSMs can be used for various purposes by means of *distributing* and *replicating* a application state.

With the aim of processing serialized requests in synchronous manner in different machines in a cluster, *replicating* a single state machine which is going to work in a *distributed* way is densely used approach of modelling for satisfying fault tolerance requirements for cloud systems. By using RSM phenomenon, *consistency* can also be satisfied in different aspects -eventual consistency or strong consistency- by major cloud vendors, as [1] proposed. Moreover, by using this approach, due to statements by [2], *high available* and *high performance* data store topology can be constructed for more-reliable cloud services. The aim of using the term *reliable* is reflecting an ability to perform required functions under given conditions.

With this in mind, checkpointing and recovery approaches are widely used for ensuring fault tolerance in different cloud services. As a result of that situation, applying different checkpointing approaches in replicated state machine systems is an highly active research topic. Basically, in the scope of thesis, breaking execution history of RSMs separate parts will be discussed in detail. In order to prove the efficiency of this type approach, ten-fold experiments with different number of replicas along with different type of solutions are carried out, results are shown in illustrative way with different charts.

DCfRSM approach may be used as checkpointing solution for the distributed computing systems, especially for the RSM systems. *DCfRSM* offers potential features in order to bring fault tolerance property into RSM systems, such as eliminating single point of failure, distributing checkpointing and recovery overhead to the all instances in the cluster and ensuring the replica consistency.

There exist industry-standard approaches for storing checkpointing artifacts inside state machine execution context. The source of inspiration behind these approaches can be summarized as:

- storing checkpoint information *outside* of the state machines
- *all* the checkpoints are persisted within *all* the state machines.

Within context of this thesis, a novel way of compromising these two approaches is proposed and examined. The novelty of this approach can be stated as: if only one component was responsible for storing checkpoint snapshots in the distributed computing system, it would be necessary to make this component *high-available* and *eliminate the single point of failure* in order to use the checkpoint snapshots needed by other replicas at any point of time. It is obvious that a failure in this component would cause loosing all the checkpoint images that has been taken so far, which expresses a need for eliminating the stated single point of failure. However, if *all* the state machines was responsible for *all* the checkpoint snapshot images, not only it would bring in higher tolerancy in the state machine replicas, but also bring in more overhead, in terms of memory and CPU utilization. By using *DCfRSM* approach, since the events - and also, checkpointing process- are *distributed* in sporadic manner, eliminating both *single point of failure* and *overhead* during persisting system state becomes more easier to handle. Moreover, whilst execution of state machines inside each replica in the distributed computing cluster system, *DCfRSM* proposes to *distribute* the checkpoint snapshots to the active member of the clusters. By doing so, whenever a failure occurs one of the instances, there will be no burden to restore the failing replica, since all the checkpoint snapshots -which includes all the states passed, all the events processed events within the context of state machine- can be gathered with ease. By using *DCfRSM* solution, the more RSM instances brings in the more fault-tolerant systems in the scope of RSM architecture. In order to prove this scenario, architectural concepts has been prepared with the aim of examining benefits of *DCfRSM* approach. *Geographically distributed servers* hired by from cloud vendors is needed for this purpose, geographically distributed RSM instances has been executed for a specific period of time and then, results are evaluated with the aim of examining prepared solution.

Within scope of this thesis, third-party tools are needed for communicating distributed RSM instances. For the inter-process communication purposes, *RabbitMQ* broker is densely used. By attaching two or more RSM instances which are geographically distributed while sending events to be executed by state machines and pass checkpointing artifacts between instances, *RabbitMQ* arose as a lightweight solution during integration to *DCfRSM* architecture. In initial status of this thesis, checkpoint snapshots are persistently stored on the filesystem of the RSM instance as a *MongoDB* document. On the other hand, due to input/output operations performed on the filesystem of RSM instance regarding database operations are produced overhead, so using *MongoDB* for storing checkpoint snapshots is left over. It is decided that checkpoint artifacts stored on *RAM* as a special structure, instead of storing on filesystem.

Moreover, in order to benchmark *DCfRSM* approach, few other solutions are implemented for comparing the results. *Conventional Solution* forces all the replicas which are alive and executing RSM to perform checkpointing after each and every event processing so that all the replicas stores identical number of checkpointing snapshots during state machine execution. During execution with *Centralized Solution*, none of RSMs store none of the checkpoints triggered due to state machine transitions. Moreover, all the checkpoint images generated by RSM instances are stored by an *external* instance, which is *LoadBalancer* instance in this case. One another point neded to be issued here is Java Spring State Machine framework and its internals -Listener, Factory, Annotation, Action- has been widely used for developing *DCfRSM* and other approaches.

Experiments are executed within both docker-compose environment and IaaS servers hired cloud vendors. In all the experiments, RSM instace memory footprint and restore duration of replicas are investigated. Results from these experiments show parallelism with each other and also states *DCfRSM* mechanism provided advantage in terms of memory footprint compared to *Centralized Solution* and *Conventional Solution*. Memory footprint of each RSM instance in experiments are measured with an external library -specifically, a jar application- which uses *Instrumentation API* that the JVM exposes for developers. Another key take-away from experiments is by using *DCfRSM* approach, memory consumption of whole cluster is highly decreased when the number

of RSM instances in the cluster are increased. Therefore, all the experiments are replayed with different number of instance combinations.

Due to results from experiments, even though there seems an advantage in using *DCfRSM* in terms of memory consumption as stated earlier, it seems *DCfRSM* produces a high recovery time when it is compared to *Mirrored*, *Centralized* and *Conventional* approaches. Since the extra communication overhead needed for collectiong partial histories from different replicas inside cluster, recovery time for *DCfRSM* approach is increased.

Rest of the thesis is organized as follows: literature review results is presented on Chapter 2. Chapter 3 discusses the background information of the *DCfRSM* solution in detail. Architectural concept, implemented solutions for benchmarking purposes, simulation environment are represented on Chapter 4. Chapter 5 presents the results of the experiments and the thesis is concluded in Chapter 6.

2. RELATED WORK

There exist numerous researches regarding the aim of evaluating the performance of checkpointing approaches within distributed computing domain, as such, there are tons of techniques to be used as a solution with the aim of minimizing checkpoint and recovery costs. As [3] states, there is need for replicated state machines to guarentee that majority of replicas inside cluster can communicate with each other and be prone to node failures, network partitions for distributed and asynchronous environments. While [4] proposes a novel replication technique with the aim of decreasing recovery costs in case of failures on physical machines, in order to minimize checkpointing costs, [5] proposes a brand-new solution by focusing storage aspects. [6] focuses on network aspects and proposes a high-performance replicated state machine checkpoint and recovery approach reproduced from Paxos consensus protocol. There are numerous efforts derived from Paxos regarding different replicated state machine optimizations. [7] proposes an efficient implementation for snapshotting and recovering current state of the state machine with including pipelining, batching and multi-threading. In order to minimize overall checkpoint overhead, [8] propose to checkpoint *only straggling tasks* in order to minimize the number of checkpoints. Within the scope of this thesis, instead of persisting checkpoints after only certain tasks as suggested by [8], it is decided to persist checkpoints to be triggered just after *every* task execution inside state machines. With the aim of reducing the checkpoint data size, [9] propose a novel checkpointing mechanism which is named as *incremental checkpointing* approach and works by modelling a decision algorithm in order to reveal the dirty pages that are modified since last checkpoint time and persist the minimal essential data to be checkpointed. Incremental checkpointing introduced by [9] is adopted in terms of *time ticks* and *execution history* in this thesis, instead of forcing all the replicas to checkpoint all the modifications performed on the internal states so far, it is decided to store events occurred within predesignated *time ticks* for a predesignated execution steps. [10] notes an concurrent replication technique for the replicated state machines with the help of static analysis methodology with the aim of compensating

the nondeterminism. [11] proposes a low-latency replication technique for replicated state machines running as geo-replicated software services which includes serialization of commands.

There are many other efforts which is aimed to find proper a way to implement an efficient checkpoint and recovery mechanism in user level [12] or kernel level [13]. [12] states that the user level checkpointing is performed explicitly by external applications and hence, user-level application is unaware whether it is checkpointed or not. [13] proposes an innovative approach called buffered co-scheduling which is implemented at kernel level, hence has unrestricted access to hardware and software resources easily so that UNIX/LINUX signal mechanism can easily be used for checkpointing formulations.

Efficient checkpointing and recovery mechanism in the context of RSM systems has already other application areas in the distributed computing domain. As stated by [14], efficient recovery execution is implemented in the presence of arbitrary faults can be stabilized. [15] proposes to use *divide-and-conquer* approach for the fault-tolerant replicated state machine cluster systems. According to [16], rather than waiting for the result of high-load -which means CPU-bound or I/O-bound operations- processes during RSM execution, a decision system inside state machine cluster may instead predict the outcome of that operation and *checkpoint* its current state, and speculatively execute subsequent modifications by using the result predicted in previous state. If the speculation is correct, then *checkpoint* is made durable and persistent, otherwise, RSM cluster rolls back to previous state to the *checkpoint* and re-execute further operations for ensuring durability. So that, *checkpointing* will be used as a milestone for the RSM execution. But a trade-off exists for this approach, it is decided as beneficial if the time interval of checkpointing is less than the time interval of performing operation which generates the expected result.

Besides checkpointing approaches, there has also been interest in *recovery* mechanisms with respect to state machine execution context. Due to [17], there are various industry-standard tools which adopts recovery approach in the context of state machines in different aspects. [17] names this approach as *declarative system update* and has roots in defining the *desired state* of system, applying necessary modifications to *current state* in order to achieve the *desired state* by using RSMs in

different context. In addition to that, [18] focuses on the recovery of failed replicas and states a novel *recovery* approach for faulty replicas by parallelizing the execution of checkpoints using *multithreading* phenomenon inside RSMs so that provides a chance for achieving consistency for both faulty and regular(non-faulty) replicas. The basis for [18] is execution of concurrent commands in the context of the state machines and proposes two modes for execution: coordinated and uncoordinated. [19] states three novel recovery approaches that are proposed as does less work and hence produces less overhead during restoring to previous state in faulty replicas in the RSM cluster. *Doing less work* and hence *producing less overhead* is also inspired from [19] for the checkpointing.

Due to review by paper [20], several layers of fault-tolerance may be defined, such as *optimistic fault-tolerance* and *conservative fault-tolerance* mechanisms which are novel approaches for distributed computing services. [20] also states that, by using *checkpointing* and redo mechanisms, there is a strong chance for ensuring replica consistency for the RSM clusters. Achieving *replica consistency* stated by [20] is one of the features proposed by the *DCfRSM* approach, as well.

The idea behind using replicated state machines in order to model distributed checkpointing approach is already stated by [2] and [21], replicated state machines can be made fault-tolerant with feeding the same inputs to multiple computers which is the approach used as fundamental principle within scope of experiments of this thesis study. Moreover, [1] states that increasing the quality of the user experience is highly dependent on making systems *replicated* across *geographically* by using replicated state machines. As suggested by [1], specific subset of experiments are also executed on cloud systems to be able to examine the advantage of *DCfRSM* approach on *geographically distributed* and *replicated* state machines. [18] also represents a efficient logging mechanism along with an efficient checkpoint model which executed in *parallel* and *distributed* manner by executing concurrent commands which is more challenging than in classic state machine replication. By using this approach, not only *recovery* process is being parellelized but also *checkpointing* is persisted concurrently in all replicas. Making checkpointing and recovery process parallelized suggested by [18] is positioned as a future work within the context of *DCfRSM* with the aim of increasing the overall performance during state machine execution.

3. DISTRIBUTED CHECKPOINTING

In today's cloud computing systems, in order to bring fault tolerance as a non functional requirement offered by major cloud vendors, checkpointing phenomenon is widely used. Especially, applying different checkpointing approaches in RSM systems is an highly active research topic. More specifically, in order to process sequential requests in synchronous manner, replicating and replaying a single logical state machine which is going to work in a distributed way is densely used way of modelling for providing fault tolerance. By using *fault tolerance* term, it is intended to reflect an ability to self-detect and correct all failures, such as hardware and software failures. As stated by [22], in the presence of failures, faults are corrected without disturbing whole execution process. In this chapter, scope and background of the *DCfRSM* will be discussed first. After that, its internals will be introduced in detail.

3.1 Background Information

As [2] stated, RSMs provide serialized and durably committed modifications as of a result is returned from the system components. As [16] stated, RSM systems provides an approach in order to tolerate numerous faults which includes hardware failures, software failures and so on. By using fault tolerance feature indicated, in this thesis, advantages of distributing snapshots of RSMs in order to decrease the memory overhead during persisting system state is investigated. This approach includes deploying and serializing of request history into many pieces during persisting checkpoints so that each RSM instance is going to store a specific piece of history instead of full execution history while overall history is going to be gathered from the overall system, which also means, logical master history will be shared between all the active state machine replicas.

The idea behind using RSMs with the aim of modelling distributed checkpointing approach is already stated by [2]. Fault tolerancy can be established for the RSMs by running same and single state machine on multiple instances with feeding same

inputs in the same order to each of them. By doing so, each and every RSM in the current cluster can be driven by fully serialized inputs, even if the freshly booting one or failing one.

In order to bring new though process around checkpointing mechanism within RSMs, it is needed to examine proposed solution with many other solutions with the aim of proving its advantages. As [12] stated, external applications may be used for checkpointing purposes which provides inspiration for designing and implementing *Centralized Solution* while benchmarking *DCfRSM*.

3.2 DCfRSM as a Checkpointing Solution

In *DCfRSM* method, internal aspects which are needed to be included in RSMs in order to synchronously store, serialize and merge logical execution history by the currently running replicas are implemented. By doing so, each of the RSM instances persists one or more pieces of the whole execution history inside its local contexts, in case of any need to be retrieved by a freshly booting replica or in case of failure by failing replica.

In order to extend *DCfRSM* method to be more redundant, some other ideas exist for to be applied. By using similarity with the RAID structure and striping protocol for the high-available storage systems, *DCfRSM* can be extended as if more than one RSMs will persist exactly same checkpoint image. There seems to be clear need for determining metrics while choosing the replica groups in order to store the checkpoints of each other during execution. One straightforward approach is *pairing* and *mapping* RSMs with each other. This idea will be discussed in detail in *Mirrored Solution* section.

It is assumed that each and every RSM instances includes some states stated as s_x, some events triggering transitions between state machine states such as $s_x \xrightarrow{a_c} s_y$ and by triggering action a, transition from state x to state y is possible in the state machine context. As stated, by using these definitions, it is possible to formulate a state machine model which is defined in 3.1. In equation 3.1, δ defines the transitions which is a function from state-action pairs to states.

$$
\begin{aligned}
&A = \{a_0, a_1, \ldots\} \\
&S = \{s_0, s_1, \ldots\} \\
&M = \{A, S\} \\
&\delta = A \times S \rightarrow A
\end{aligned}
\tag{3.1}
$$

Due to the definition in equation 3.1, whole execution history for the RSM instance can be illustrated as in definition in equation 3.2. By using this equation, it is possible to state the history begins with a initial state, execution of state machine continues by triggering regarding events which is need to achieve a desired state. This expression is beneficial when there is a need to represent parts of execution history.

$$
H = s_x, a_c, s_y, a_d, s_z, a_e, s_t \ldots \tag{3.2}
$$

Equation 3.2 includes sequence of state transitions. The aim for using superscript is representing a discrete clock tick interval which begins from *tick 0* as the beginning time for the whole execution history. The aim for using subscript associating a history with the id of a RSM instance which currently checkpoints the given part of the whole execution history.

$$
\begin{aligned}
&H^{0-79} = s_0, a_0, s_1, a_1, s_2, a_2, \ldots, s_{39}, a_{39}, s_{40} \\
&H_0^{0-19} = s_0, a_0, s_1, a_1, \ldots, a_9, s_{10} \\
&H_1^{20-39} = s_{10}, a_{10}, s_{11}, a_{11}, \ldots, a_{19}, s_{20} \\
&H_2^{40-59} = s_{20}, a_{20}, s_{21}, a_{21}, \ldots, a_{29}, s_{30} \\
&H_3^{60-79} = s_{30}, a_{30}, s_{31}, a_{31}, \ldots, a_{39}, s_{40}
\end{aligned}
\tag{3.3}
$$

As indicated in equation 3.3, whole execution history denoted as H^{0-79} while between time 0 and 79 ticks, 40 events are triggered and 41 different states has been travelled. Moreover, equation 3.3 also indicates that there exist four unique RSM instances denoted by i as replica id in the current cluster for storing full execution history denoted as H_i^{s-e}, while s and e represents beginning and end time tick. Equation 3.3 also represents that the full execution history is divided into four different separate histories while each part is persisted in a different state machine context. This definition also

expresses that, H^{0-79} denotes the single logical master history which is shared and distributed among all the RSM instances.

In order to present a concrete example regarding equation 3.3, it is supposed that a distributed computing cluster system has been organized with the aim of processing requests from several clients. It's also assumed that all the instances in the cluster system exposes several API endpoints for public usage in order to execute some operations on the given data in the request and hence, provides opportunity to modify system state. By processing requests coming from clients, instances also executes one *identical* and *logical* state machine and as such, distributed RSM cluster can be constructed from regarding instances. Clients simply sends HTTP requests to the RSM instances in order to modify the system state. Another assumption is, all the RSM instances in RSM cluster are running as IaaS instances which are hired from a major cloud vendor. Moreover, stated instances located on *different* server farms on the *different* regions which is provided by cloud vendor. It is decided that 4 instances will be enough for visualizing *DCfRSM* concept. The RSM instances in distributed computing cluster is located in this way:

- RSM_i is located on the region Europe
- RSM_{i+1} is located on the region Middle East
- RSM_{i+2} is located on the region Canada
- RSM_{i+3} is located on the region Asia

Once the RSM cluster is in up and running state, it means that only one *logical* and *identical* state machine will be executed by all the RSM instances. In order to ease the statement, only one client will be considered which acts as HTTP client and sends HTTP requests to RSM instances in the cluster. Due to equation 3.2, 79 ticks will be passed during the specified time interval and during this interval, several requests will be sent to RSM cluster by HTTP client. Between time ticks 0 and 19, RSM_i -region Europe- will be storing checkpoints artifacts regarding RSM execution while between time 20 and 39 is persisted by RSM_{i+1} -region Middle East-, between time 40 and 59 is persisted by RSM_{i+2} -region Canada- and between time 60 and 79 is persisted by RSM_{i+3} -region Asia-. In contrast to explanation above, if all the checkpoints was

stored by only one RSM instance or on a separate IaaS instance which is used as a temporal storage area, it would not be possible to gather checkpoints in case of any failure occurs on that RSM instance, since this instance is a *single point of failure* in the cluster system and needs to be eliminated. By using *DCfRSM* approach, failing instance can be recovered with the help of other health -non-failing- RSM instances in the cluster and hence, distributed computing systems will have a liberty to establish a fault tolerance with ease.

4. SYSTEM ARCHITECTURE AND EXPERIMENTAL ENVIRONMENT

This chapter describes the architectural decisions, considerations regarding simulation and experimental environment while benchmarking *DCfRSM* solution.

4.1 Overall Architecture

4.1.1 Development environment and development artifacts

In order to compile and generate artifacts from source code repository as a whole, a build system is needed. During development of *DCfRSM* solution and other correlated solutions, *Maven* build system is used in order to get, build, extract and use necessary dependencies -MongoDB driver, Apache Zookeeper, AQMP, log4j- along with *Java Spring State Machine* framework. With this in mind, for development environment, *IntelliJ IDEA* has been used for developing source code and generating artifacts -JAR file for RSM execution- as IDE because of having wide support and integration options for *Maven* build system. In addition to that, *Java Spring Boot* framework was also integrated into source code environment in order to speed up and simplify build configuration, dependency injection and hence, development process.

4.1.2 Context of state machines

In this section, states and state transitions of state machines, features that are utilized from *Java Spring State Machine* framework will be discussed in detail.

4.1.2.1 States and state transitions

In the cluster environment containing RSM instances, states of the state machine in all replicas exercises the states of a book in the simple book store application, as shown on Figure 4.1. When a book is ready to be bought from customers, it starts with *UNPAID* state and waits the *PAY* event to be triggered, as guessed. When the booking and payment operations are performed on the book, *PAY* event is triggered and state has

been changed from *UNPAID* to *WAITING* state. In this state, book store *waits* the receipt from customer in order to ship the book. Once the receipt is received by book store, *RECEIVE* event is triggered and state is transited from *WAITING* to *DONE*.

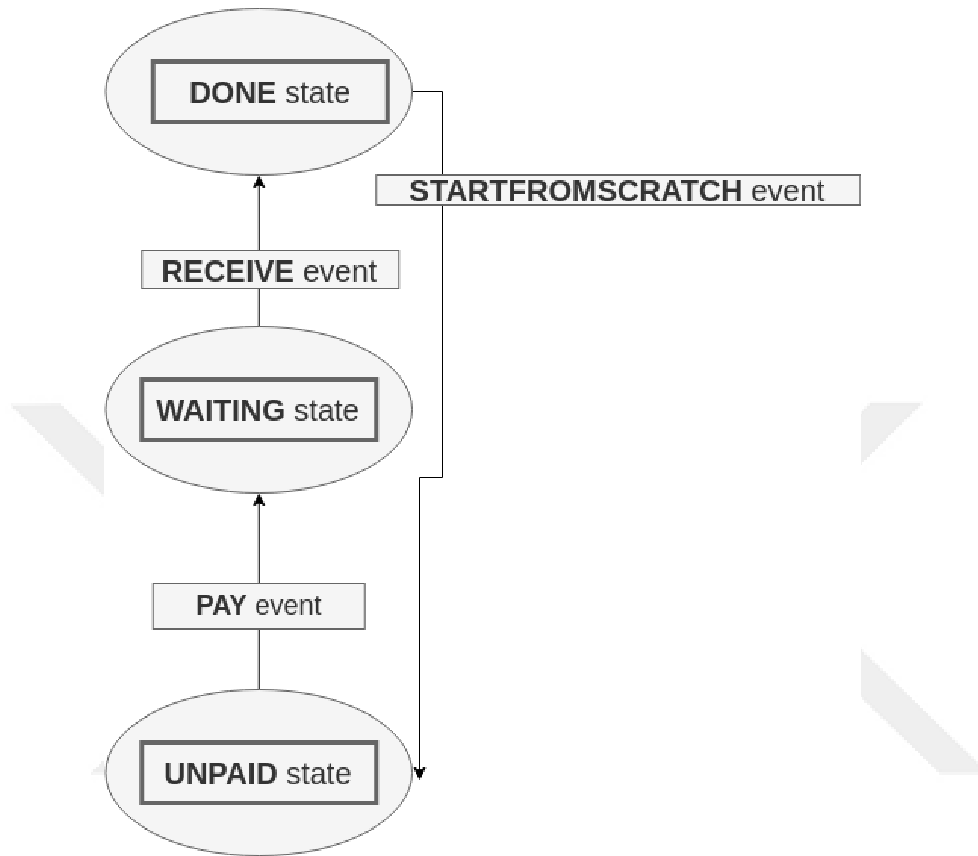

Figure 4.1 : State transitions.

As illustrated on Figure 4.2, RSMs execute basic modifications on its own local contexts, such as incrementing and decrementing local variables once an event is started to be processed inside state machines, and also updates shared variables while transiting from one state to another. The purpose of using local variables inside state machine states is measuring how many times the currently executed state is executed beforehand and as such, calculating how many times state machine is passed through the same state in the time of execution. On the other hand, the purpose of using shared variables between state machines states is to be ensure about all the RSM instances in current cluster will reveal same state -and hence, same value for the shared variable- in any time during state machine execution.

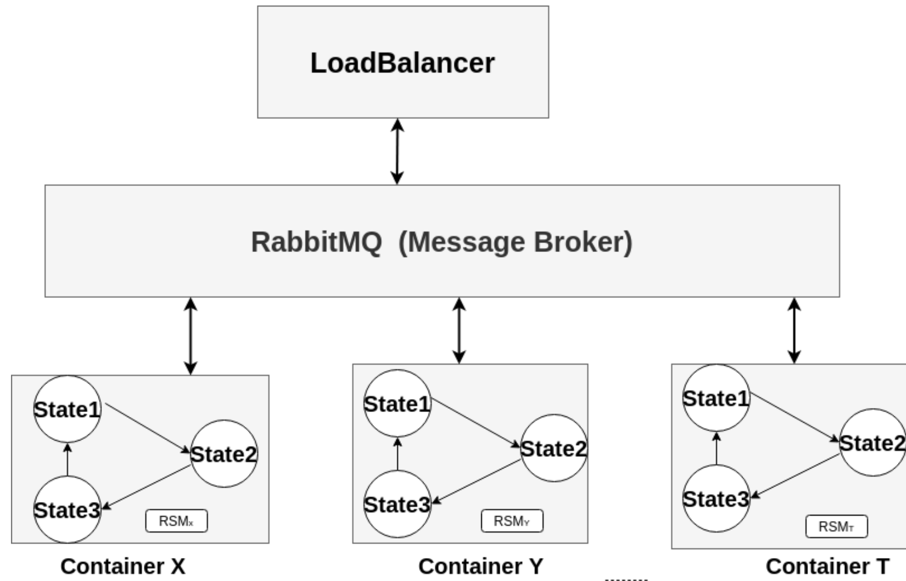

Figure 4.2 : Overall architecture.

4.1.2.2 Actions

Thanks to the features of *Java Spring State Machine* framework, some *actions* can be initialized which are ready-to-executed with state transitions and states. By using the *action* feature, a group of operations are associated with the transition from $UNPAID$ to $WAITING$ or from $WAITING$ to $DONE$. Basically, in the scope of this thesis, *shared variables* which is explained before and intended to express the current state of state machine is modified as a *transition action*. Moreover, *entry actions* and *exit actions* are defined in order to modify and report local variables inside states $UNPAID$, $WAITING$ or $DONE$. As a convention, local variables are updated during *entry actions* of each state and inside *exit actions*, current status of local variables are reported in order to report current status of state machine execution context.

4.1.2.3 Listeners

In order to *listen* state transitions, react events and get callback when various changes inside of state machine context, *Listener* component are configured and densely used within scope of this thesis study. Once the RSM instance is started to initialize, listener/listeners are directly attached to state machine environment so that any change -state transitions, failures- during execution can be handled and necessary action can be taken regarding changes. For instance, when a software crash occurs in one of the

state machines, *Catcher* functions are triggered which are defined in *Listener* class in order to report what went wrong.

4.1.2.4 Factories

Factory configuration option is one of the advanced interfaces that Java Spring State Machine framework exposes for software developers. Aim for using factory configuration option is creating dynamic RSM instances during *compile time* instead of *runtime*. This functionality can be tought as *a set of configuration options* which are available for state machines that are ready-to-use during state machine initialization phase. By using *Factory* functionality, initial states of state machines and other configuration options are supplied programmatically.

4.1.2.5 Annotations

Meaning of usage of *annotations* is configuring lifecycle methods which are executed once the state machine are *started -OnStateMachineStart* annotation- and *stopped -OnStateMachineStop* annotation-. By using annotations, callbacks are configured for RSM initialization and RSM destruction phase which are stands for starting/stopping application services and preparing/deleting configuration files inside runtime environment.

4.1.3 Execution of state machines

Once all the replicated state machines inside current cluster are started, they prepares themself to process events -initialize local and shared variables and so on-, and then, enters to sleeping state till any event triggered in order to perform transitions between state machine states. *LoadBalancer* node, which is shown on Figure 4.2, creates necessary events and triggers them to executed by replicated state machines, when the events are ready, LoadBalancer node forwards events to the replicas via *RabbitMQ* broker. LoadBalancer node is responsible for triggering events and choosing the first RSM instance to be active in current turn. Once the first event is forwarded to any of the state machines, it processes this event, performs necessary operations on its local variables, updates shared variables which is shared among states and finishes its execution in order to process a new event. After first event is processed welcoming state machine, regarding event is distributed to all the replicas inside the cluster in order

to be ensure about maintaining a *replicated* state machine and hence, all the replicas will maintain same state after each event processing. The synchronization between clients is most critical point of sustaining more than one replicated state machines inside a cluster.

LoadBalancer component has different working modes while triggering events for the state machines. While choosing the welcoming RSM instance which contains the first state machine that executes the event entering the cluster, *randomized* mode or *ordered* mode can be chosen. If the regarding state machines uses *Conventional Solution* for checkpointing purposes, LoadBalancer will be in *ordered* mode and chooses the welcoming state machine in ordered way, can be summarized as by skipping *checkpointing* steps:

- LoadBalancer produces event e_i, forwards to state machine RSM_1
- Once the state is transited on RSM_1, event e_i is broadcasted to all the replicas in ordered way -RSM_2, RSM_3 and so on-, so that all replicas will maintain the same state
- LoadBalancer produces event e_{i+1}, forwards to state machine RSM_1
- Once the state is transited on RSM_1, event e_{i+1} is broadcasted to all the replicas in ordered way -RSM_2, RSM_3 and so on-, so that all replicas will maintain the same state

However, if *LoadBalancer* is configured with *randomized* way, that means the state machines are configured with *DCfRSM*, *Centralized* or *Mirrored* approch for checkpointing purposes. After constructing message for event processing, *LoadBalancer* chooses welcoming state machine in random way, which can be illustrated by skipping *checkpointing* steps:

- LoadBalancer produces event e_i, forwards to state machine RSM_4
- Once the state is transited on RSM_4, event e_i is broadcasted to all the replicas inside cluster -RSM_1, RSM_2 and RSM_3-, so that all replicas will maintain the same state
- LoadBalancer produces event e_{i+1}, forwards to state machine RSM_3

- Once the state is transited on RSM_3, event e_{i+1} is broadcasted to all the replicas inside cluster -RSM_1, RSM_2 and RSM_4-, so that all replicas will maintain the same state
- LoadBalancer produces event e_{i+2}, forwards to state machine RSM_1
- Once the state is transited on RSM_1, event e_{i+2} is broadcasted to all the replicas inside cluster -RSM_2, RSM_3 and RSM_4-, so that all replicas will maintain the same state

4.1.4 Checkpoint artifacts

After event processing steps are finished, that's where checkpointing process comes into play. During life cycle of replicated state machines, checkpointing occurs over and over again. As per the checkpointing approach to be chosen which will be explained in *Implemented Solutions* section, first milestone is whether the checkpoint will be stored by replicated state machines or by external application which is also stated by [12], in our case, LoadBalancer node will be responsible for storing checkpoints as external aplication. Second critical milestone is determining how many replicas will store checkpoint artifacts regarding processed event a little while ago. The answer should be 1 for the *Distributed Solution*, 2 for the *Mirrod Solution* or *all* for the *Conventional Solution*. While checkpointing the state, *context* of the state machine is stored as checkpoint artifact.

It is assumed that during processing event e, context of the state machine are denoted as C_e :

$$C_e = I_e + O_e \tag{4.1}$$

Due to 4.1, context consists of *inputs* of states which is denoted as I_e and *outputs* of states which is denoted as O_e. Moreover, context of the state machine are stored as a whole. Inputs are consist of incoming event, event timestamp, source state of the state machine while outputs are described as local and shared variables, destination state of the state machine. As a result of checkpointing process in each replica, whole execution history is recorded in splitted manner in different replicas executing replicated state machines. Once the checkpointing artifacts are ready to store, it

persisted on memory within a specific structure which is designed and implemented for this purpose. At initial steps of this thesis study, it is decided to store checkpoints objects in a *MongoDB* database collection, but due to overhead of querying checkpoint objects inside databases, it is decided to store checkpoint objects on memory instead of databases, as also indicated by [2].

4.2 Interprocess Communication

As can be seen from Figure 4.2, *RabbitMQ* is located as message broker among all the RSM instances and *LoadBalancer* node. The usage of *RabbitMQ* broker can be summarized as:

- Once an event is triggered by *LoadBalancer* to in order to processed by RSM instances
- When RSM instance is finished processing, informing *LoadBalancer* with acknowledge message
- As stated on *Centralized Solution*, by-passing checkpoint information from RSM instance with the aim of storing by *LoadBalancer* node
- Measuring and reporting number of messages passed through during event processing

Once an replicated state machine is suddenly stopped during event processing or during state transitions, it means that the state machine is *failed* needs to be *restored* and re-joined to cluster again. If this is the case, execution history can be gathered from non-faulty clients via a broadcast message by the failing replica and then, execution history needs to be *serialized* in the order of execution. After serializing and applying checkpoints to the failing replica, it is ready to re-join to existing cluster again. As stated earlier, all the messages needed to recover failing replica passed through *RabbitMQ*, measure and reported in continuous manner.

4.3 Implemented Solutions

4.3.1 Proof of concept study

At initial steps of this thesis study, with the aim of using *Distributed State* property of the Java State Machine Framework, *ZK* instance was used to be located as cluster communication medium in order to store whole execution history. By doing so, the aim was using *distributing states through a state machines* inside cluster environment. In order to include desired functionality, RSM instances needed to be configured in a pre-designed way. Once *Distributed States* are configured, whenever a RSM processes event triggered by an action, necessary checkpoint information was being stored on *ZK* internal file system. As a result of this storage phenomenon, freshly booting replicas and failing replicas needed to connect to ZK instance at their initialization/re-initialization phase, gather checkpoint snapshots and apply them in order to join/re-join the current replica cluster. It is obvious that, this situation makes ZK system a single point of failure, whenever a failure occurs in ZK, whole RSM cluster loses the checkpoint images that has been taken so far. In order to eliminate the hassle of dealing with constructing a high-available *ZK* cluster, using ZK instance with distributed states is left over, instead of this functionality, some solutions are implemented for benchmarking purposes. In addition to that disadvantage, one additional component was needed to be included in cluster environment so that the need for maintainability and durability of whole cluster was increased when ZK instance along with *Distributed States* property was chosen. By leaving out *ZK*, more maintainable RSM cluster environment is being prepared.

4.3.2 Conventional solution

One of the checkpointing approaches that is used for benchmarking *DCfRSM* solution is conventional checkpointing. As can be seen on B.1, along with this solution, all the replicas which are alive and executing RSM does perform checkpointing after each and every event processing. By doing so, after processing the state transition triggered by an event on any of the replicas in the cluster, not only all the replicas will set the same state, but also all of the replicas will checkpoint exactly same data and will be

storing exactly same number of checkpoints. In case of any failure on any of the RSM instances, all the checkpoint information should be gathered and applied in order to re-join the failing cluster again. Same situation exists for freshly booting replicas, once the RSM is initialized for the fresh replica, there is need to communicate with the replicas inside the cluster in order gather checkpoint information that has been taken so far. Since all the replicas stores *same* and *all* the checkpoint sequence, one of the active RSM instances can be chosen for gathering all the checkpoints that has been taken so far. Due to situation stated on previous sentence, memory overhead of this approach is expected to be more larger than other solutions, since all the replicas stores exactly *same* and *all* the checkpoint snapshots.

4.3.3 Centralized solution

In case of centralized approach is chosen as checkpointing solution, none of RSMs store none of the checkpoints triggered due to state machine transitions. After processing the event triggered which also makes replicated state machines to transit one state to another, no checkpointing process is driven by the replicated state machines and then, all of the clients will be in pending state while waiting to process next event. In other words, no checkpoint will be stored by the alive replicas no matter how many events processed by replicated state machines inside cluster. Besides this, all the checkpoint messages are stored by an external application in order to ensure about all of the events processed by all the replicas is persisted in permanent manner.

Flow can be summarized as below:

- *LoadBalancer* node creates event e_n to be processed RSMs, and then, forwards this event to RSM_4
- Once the state is transited on RSM_4, event e_n is broadcasted to all the replicas inside cluster, so that all replicas will maintain the same state
- Checkpoint message is prepared by RSM_4 and sent over *RabbitMQ* to *LoadBalancer* node with an acknowledge message indicating all the RSMs inside cluster finished processing event e_n, now ready to process for e_m
- *LoadBalancer* node stores checkpoint information for e_n and then, creates event e_m to be processed RSMs, and then, forwards this event to RSM_2

In case of any failure occurs one of the live replicas in the cluster, persisted checkpoints are gathered from external application -which is *LoadBalancer* node, as stated earlier- by the failing -as of now, recovering- replica.

4.3.4 DCfRSM

Once the *DCfRSM* is chosen as fault tolerance algorithm, the RSM instance which processes first event coming into cluster system is responsible for storing checkpoint information regarding this event. To be more clear, once an event is welcomed by any of the RSMs inside the cluster, checkpoint regarding this event is stored locally by the state machine which processed this event on its local context. As a result of this approach, each of the state machines does not store whole execution history -which includes state transitions, states, local and shared variables-, they only store minimal portion of the execution history on its local context.

Flow can be summarized as below and illustrated on B.2:

- *LoadBalancer* node creates event e_n to be processed RSMs, and then, forwards this event to RSM_7
- Once the state is transited on RSM_7, event is broadcasted to all the replicas inside cluster, so that all replicas will maintain the same state
- Checkpoint is persisted by RSM_7
- RSM_7 informs *LoadBalancer* in order to acknowledge all the RSMs inside cluster finished processing event e_n, now ready to process for e_m
- *LoadBalancer* node creates event e_m to be processed RSMs, and then, forwards this event to RSM_4

During a freshly booting RSM instance joining replica, it does gather all the checkpoint snapshots from all the active replicas in the current cluster and hence, communication overhead is for gathering checkpoint snapshots exist for this type of solution.

In a nutshell, *DCfRSM* utilizes the discrete time depending on executing actions in replicated state machines with the aim of partitioning checkpoint persistence responsibility between current members of cluster.

4.3.5 Mirrored solution

This approach is the derivative of the *DCfRSM* solution. This solution utilizes the idea of implementing *striping* solution which is highly used for designing RAID systems in order to construct high available storage topology. After event is processed by the responsible replica and state machine transits from one state to another, more than one RSM instance persists their current snapshots and hence, increases redundancy inside current replica group. In other words, RSM instances are paired with a group of two machines and mapped each other, so that checkpoints are persisted in synchronous manner in each group of RSM instances. Mapping is pretty straightforward and can be visualized as couples of RSM_1 and RSM_2, RSM_3 and RSM_4, RSM_5 and RSM_6, RSM_7 and RSM_8 and so on.

The listing above expresses that, once the first event is handled by the RSM_5 and checkpointed in the context of RSM_5, RSM_6 is informed to persist same checkpoint on its own local context, and vice versa. During a freshly booting RSM instance joining replica, it does not gather checkpoint snapshot from all the active replicas in the current cluster, it only communicates with the one and only one RSM instance in each pair and gathers checkpoint information from one and only one RSM in the pair. More specifically, while the RSM_9 joining to replica group, it reads checkpoint information from only RSM_1, RSM_3, RSM_5 and RSM_7. This flow can also be tracked by the B.3. Hence the communication need among RSM instances are decreased when it is compared to *DCfRSM* approach, recovery time is expected to be decreased.

4.4 Simulation Environment

4.4.1 Infrastructure

Each of replicated state machines runs as lightweight linux containers in *Docker Engine*. By using docker containers, an isolated network topology can be constructed for the cluster containing replicated state machines, and moreover, connections between replicated state machines are highly simplified. Our proposed mechanism do not necessarily need to be run inside container framework, this approach can easily be extended to any framework including replicated state machine implementation. Moreover, all the replicated state machine inside cluster can be started up in repetition

by using minimal hardware resources of the host machine, thanks to docker daemon. Within this thesis, docker commands which is used to construct the RSM instances can be seen on Figure 4.3.

```
MAINTAINER celikelozdinc "celikelni@itu.edu.tr"

ARG JDK_VERSION=openjdk8

RUN \
  apk update && \
  apk add --no-cache bash && \
  apk add --no-cache busybox-extras && \
  apk add --no-cache openrc

 # install java #
RUN \
  apk add --no-cache ${JDK_VERSION}

# Spring Boot #
ARG SPRINGBOOT_HOME=/opt/statemachineapp
COPY out/artifacts/DistributedStateMachine_jar/ $SPRINGBOOT_HOME/DistributedStateMachine_jar/
# Copy reporting tools #
COPY smoc/report.sh  $SPRINGBOOT_HOME/
COPY smoc/ObjectSizeFetcher.jar $SPRINGBOOT_HOME/
RUN chmod +x $SPRINGBOOT_HOME/report.sh
# Change Directory #
WORKDIR $SPRINGBOOT_HOME
# Copy entrypoint script #
COPY smoc/start_services.sh  $SPRINGBOOT_HOME/
RUN chmod +x $SPRINGBOOT_HOME/start_services.sh
CMD ["/opt/statemachineapp/start_services.sh"]
```

Figure 4.3 : Docker instructions.

Due to Figure 4.3, when the linux container up and ready to deploy RSM instance, it has already some reporting tools and scripts in order to measure restore duration and memory footprint during checkpointing operation. *ObjectSizeFetcher.jar* will be deployed with RSM application and binded to it as can be seen from 4.3, so that calculation during experiments is performed in an easy way. The aim of *report.sh* executable is *reporting* of current memory footprint once the current run of the experiment is finished, this script will be executed from external environment which hosts docker daemon in order to get results from cluster containing all RSMs. *openjdk* is chosen as JDK environment for studies performed during this thesis. In order to start RSM instance which is deployed as *jar* file along with *ObjectSizeFetcher* library inside JVM, *javaagent* option is used, and then, output of RSM execution is redirected

into a log file in order *search* some keywords such as *Current Memory Usage*, *Started JVM in xx seconds*, etc. for reporting purposes:

```
/usr/bin/java \
    -javaagent:ObjectSizeFetcher.jar \
    -Dtimesleep=1000 \
    -jar DistributedStateMachine.jar > log
```

4.4.2 Orchestration and automation

For the orchestration purposes, *docker-compose* tool is heavily used in order to create, start and stop containers -and hence, replicated state machines inside containers as well- inside the cluster with ease. Basically, whole application cluster including all containers -and hence, all the replicated state machines- can be started/stopped/restarted via one simple command, thanks to features of the docker-compose tool.

Another feature of docker-compose used in this thesis is *healthcheck* instruction. By using healthcheck instruction, testing all replicas inside cluster to check whether they are still functional or not is a no-brainer. All of the containers runs same query within specified frequencies and responses are constantly being flushed out to terminal by executing command *docker ps*. Due to healthcheck instruction on Figure 4.4, docker daemon executes the command specified in the *test* section in each time interval which is indicated in the *interval* section and waits for the result in *timeout* time. Due the result of this test, health state of container is determined.

```
stdin_open: true
healthcheck:
  test: echo 'db.runCommand("ping").ok' | mongo localhost:27017/test --quiet
  interval: 10s
  timeout: 10s
  retries: 5
```

Figure 4.4 : Healtcheck.

In order to automatize the booting processes of RSM instance with different configurations, *.env* files are used, as a great feature of docker-compose tool. By using it, all the necessary information regarding RSM, for example *hostname* of the linux container which executes RSM instance, configuration parameters regarding *RabbitMQ* connection -such as Queue and Exchange declarations- and iteration

number of experiments -since ten-fold experiments are used- can be passed to linux containers as a runtime property. By passing the iteration number during execution of experiments, each experiment and regarding results are numbered and ordered, so that cumulative results of all experiments can be calculated in one shot by a simple shell script.

The last but not least feature which is heavily used during experiments is constructing an isolated network structure by specifying subnets and attaching the containers executing replicated state machines into specified network. As stated on Figure 4.6, by using isolated network environment, the communication medium among containers executing replicated state machines are established as well as the connections between replicated state machines are secured. Due to Figure 4.5, RSM_{12} can be booted up via joining the network stated as *distributedWan* which is created beforehand.

```
smoc12:
  build:
    context: .
    dockerfile: smoc/Dockerfile
  networks:
    - distributedWan
  hostname:  ${HOSTNAME_SMOC12}
  environment:
    - EXCHANGE=${IPC_EXCHANGE_FOR_SMOC12}
    - QUEUE=${IPC_QUEUE_FOR_SMOC12}
    - EVENT_EXCHANGE=${EVENT_EXCHANGE_FOR_SMOC12}
    - EVENT_QUEUE=${EVENT_QUEUE_FOR_SMOC12}
    - LB_EXCHANGE=${LB_EXCHANGE_FOR_SMOC12}
    - EXPERIMENT=${EXPERIMENT}
  tty: true
  stdin_open: true
  healthcheck:
    test: echo 'db.runCommand("ping").ok' | mongo localhost:27017/test --quiet
    interval: 10s
    timeout: 10s
    retries: 5
```

Figure 4.5 : docker-compose.yaml and network instructions.

```yaml
networks:
  isolatedNetwork:
    ipam:
      driver: default
      config:
        - subnet: '192.17.41.0/24'
```

Figure 4.6 : Network instructions.

Docker-compose indicates that commands that are needed to take heartbeat from the containers are executed in specified time interval in Figure 4.4 and all the containers in up and running state. Otherwise, the state that will be shown was *unhealthy*.

For the executions of tests, debian-based machine is used for conducting our experiments, which is equipped with 2.60 GHz Intel i5 CPU, 4 GB RAM and 100 GB SSDs.

5. EXPERIMENTS AND EVALUATION

5.1 Test Bed

As shown on Table 5.1, all the experiments are conducted with 4, 6, 8 and 10 replicas in the experimental environment as described in Figure 4.2, during these experiments average amount of memory consumption used by all replicas in the cluster is measured as well as average of restore duration of newly joining replica is also measured. In order to measure memory consumption of each replica during state machine execution, an external library is used for counting number of checkpoint objects, state machine internal objects, etc. A library call is executed whenever a checkpoint is persisted during state machine execution in each of the clients and hence, current memory usage can be logged for every replicated state machine instance. This library uses *Instrumentation API* that the JVM provides for development purposes, named as *ObjectSizeFetcher* and can be inspected from Figure 5.1. By using *Instrumentation API* features, attach to a running application is allowed, which means *ObjectSizeFetcher.jar* can be attached to running RSM instances inside JVM, as can bee seen Docker instructions from Figure 4.3.

```
package itu.distributed;

import java.lang.instrument.Instrumentation;

public class ObjectSizeFetcher {

    private static Instrumentation instrumentation;

    public static void premain(String args, Instrumentation inst) { instrumentation = inst; }

    public static long getObjectSize(Object o) { return instrumentation.getObjectSize(o); }


}
```

Figure 5.1 : ObjectSizeFetcher library.

Table 5.1 : Experiments

Number of events processed	Number of replicas in cluster	Checkpoint Approach
3600	4	Conventional
3600	4	Centralized
3600	4	Distributed
3600	4	Mirrored
3600	6	Conventional
3600	6	Centralized
3600	6	Distributed
3600	6	Mirrored
3600	8	Conventional
3600	8	Centralized
3600	8	Distributed
3600	8	Mirrored
3600	10	Conventional
3600	10	Centralized
3600	10	Distributed
3600	10	Mirrored

On Table 5.2, it is presented that the restore duration for all of the checkpointing approaches for a history with 3600 requests respectively. Ten-fold input is used in terms of number of requests in order to observe the effect of checkpointing approach on restore duration.

Moreover, with the aim of extending test bed, all the experiments are also executed on the servers provided by a cloud vendor. The aim for replaying same experiments in the cloud environment is visualizing the system behaviour in a *real* environment, especially in a *geographically distributed* architecture and proving *DCfRSM* is suitable approach for checkpointing in RSM clusters. As per Figure 5.2, in order to prove the statement in previous sentence, different servers from geographically distributed regions are used for executing RSM instances. 6 virtual machines from 3 different regions are used for executing experiments in cloud environment. Due to time constraints, three fold experiments are performed on cloud environment are their results are presented. While executing experiments, *LoadBalancer* instance *RabbitMQ* broker service is isolated and positioned on a different region which is totally apart from RSM instances. All the *healthy* RSM instances distributed among 4 virtual machines and hence, spread to 2 regions. Freshly booting replicas are joint to cluster from the region which is the one apart from the regions of RSM instances, which

Table 5.2 : Average seconds of restore durations regarding 4,6,8 and 10 replicas

Distributed	Centralized	Conventional	Mirrored
5.64	4.68	5.15	5.28
6.07	4.78	5.15	5.36
6.19	4.80	5.13	5.98
6.73	5.19	5.91	6.04

means the same region with *LoadBalancer* instance *RabbitMQ* broker service. By doing so, whenever a brand-new replica joining the cluster, it is needed to gather checkpoint snapshots from *different* machines on *geographically distributed* regions on the world. As a result of this testing approach, experiments needed for measuring memory footprint and restore duration is conducted with ease.

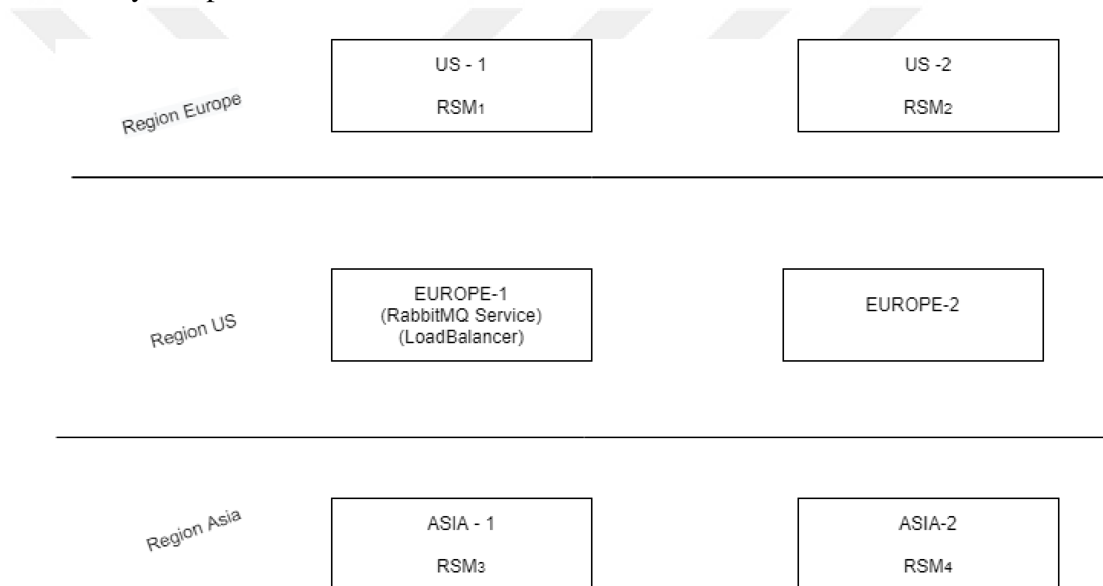

Figure 5.2 : RSM instances in cloud for the test bed with 4 replicas.

5.2 Results

In Figure 5.3, average of the memory consumption of the whole replica by using *DCfRSM*, *Centralized Solution*, *Conventional Solution* and *Mirrored Solution* are presented. Each bar on the Figure 5.3 represents the average memory consumption which is calculated by the results gathered by using external *ObjectSizeFetcher* library visualized in Figure 5.1.

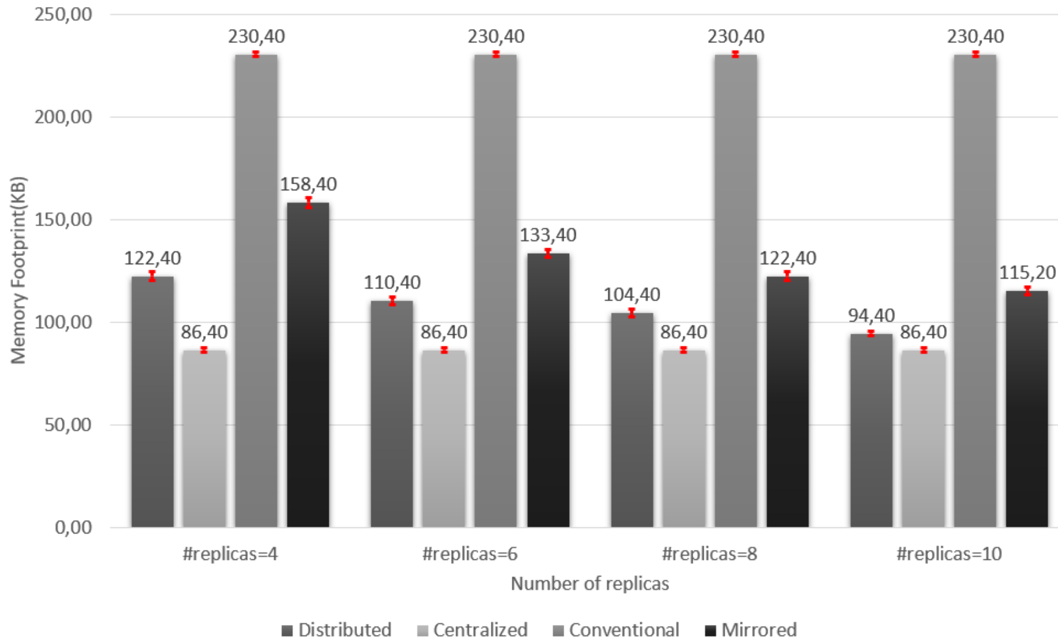

Figure 5.3 : Memory footprint consumption.

Experiments whose results are shown in this chapter are conducted in this way and recurred ten times with respective to 4, 6, 8 and 10 replicas as also visualized on Table 5.1:

- Initialize n replica in the cluster, wait for the RSM_n to be ready for processing events
- Initialize *LoadBalancer* node as illustrated on Figure 4.2
- Trigger 3600 messages in sequential mode from *LoadBalancer* node, wait for RSM_n node to finish execution
- Once all the events are processed, report local and shared variables of the all RSMs in order to ensure about *replicated state machines* are maintained in the cluster
- Boot RSM_{n+1} in order to join the cluster, wait for gathering checkpoint information from respective node/nodes

Once RSM_{n+1} finished its execution, it means that first round of the experiments are finished. As of all the experiments for the respective replica set is finished, reports can be generated. By using the flow above, total memory consumption of the cluster is calculated by excluding RSM_{n+1} and by summing memory consumption of n replica. Then, average of all ten *replica memory footprint* are calculated.

In Figure 5.4, restore duration of the freshly booting replica that is being restored with *DCfRSM*, *Centralized Solution*, *Conventional Solution* and *Mirrored Solution* are presented. In contrast to that calculation, in order to measure the restore duration of freshly booting replicas, replica RSM_{n+1} is taken into consideration. As shown on the Figure 5.5,once RSM_{n+1} finished its execution, time needed for all tasks conducted by RSM_{n+1} is measured, which are *JVM Initialization*, *Gathering Checkpoints*, *Serializing Checkpoints*, *Applying Checkpoints*.

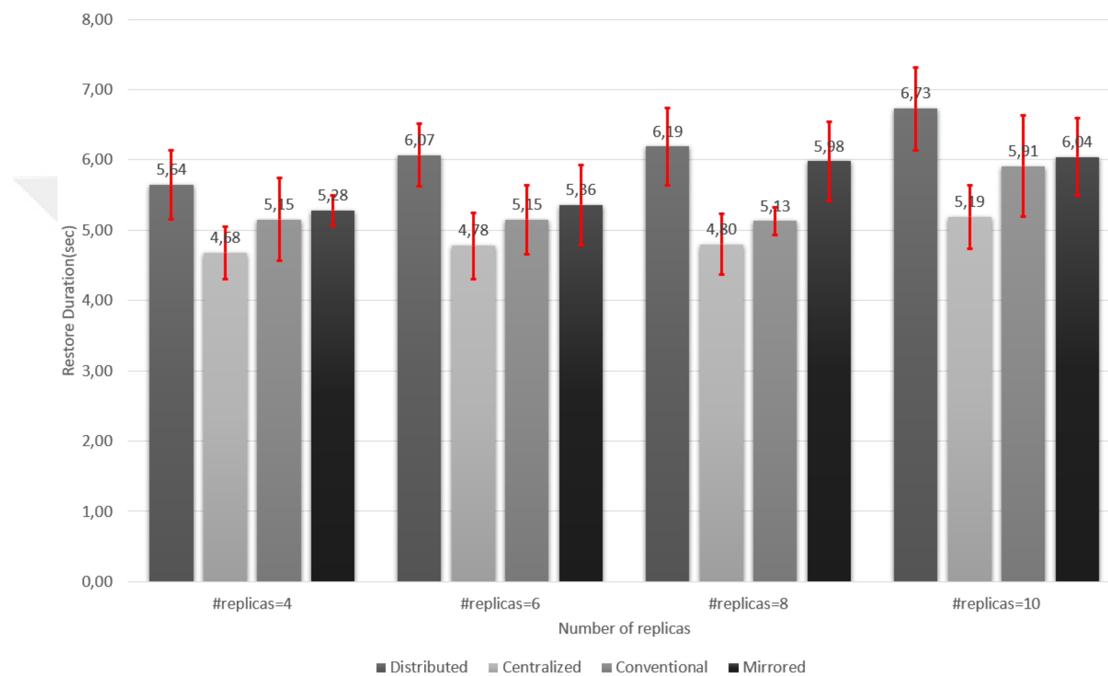

Figure 5.4 : Restore duration.

Figure 5.5 also shows us breakdown for the restore duration with respect to each of the solutions which are being explained in detail for now. Since the *DCfRSM* and *Mirrored Solution* needs to collect checkpoint snapshots from different RSMs in the cluster, one extra step is required for *serializing* checkpoint information in order to have a only one master execution history and be consistent on their contexts, which is *Serialize Checkpoints*. Due to Figure 5.5, during *JVM Initialization* phase, JVM environment for executing state machines instances are being prepared and then, state machines are being started, local and shared variables for the state machines are initialized. On *Gather Checkpoints* phase, freshly booting replica communicates with other state machines over *RabbitMQ* broker inside current cluster and receives checkpoint information which will be *applied* soon during *Apply Checkpoints* phase.

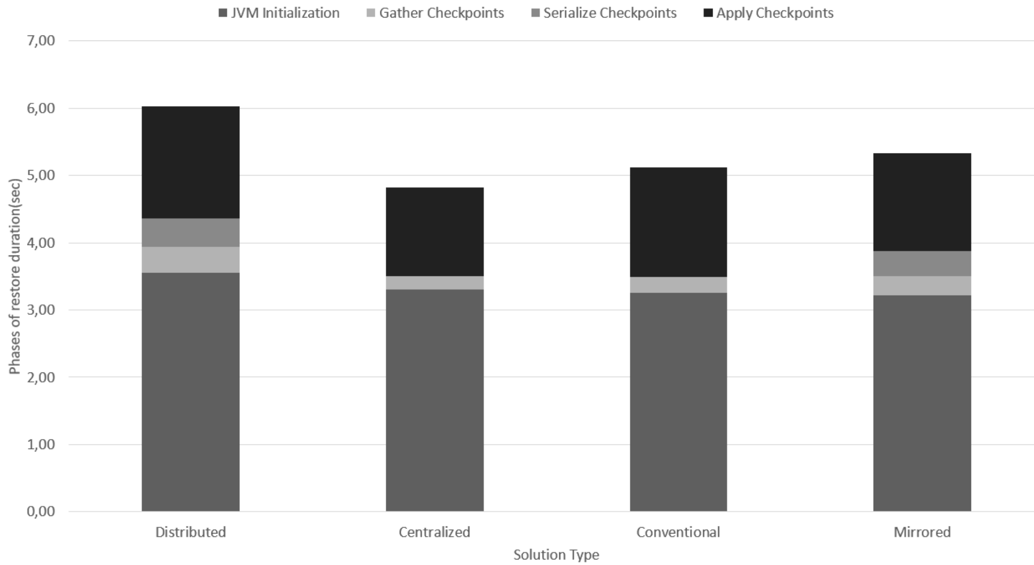

Figure 5.5 : Phases for restore duration.

5.3 Evaluation

According to Figure 5.3, *DCfRSM* mechanism provided around 108 kilobytes of advantage and *Mirrored Solution* provided around 72 kilobytes of advantage compared to *Centralized Solution* during experiments conducted for cluster containing 4 replicas. For cluster with 6 replicas, *DCfRSM* mechanism provided around 120 kilobytes of advantage and *Mirrored Solution* provided around 96 kilobytes of advantage compared to *Centralized Solution*. For cluster with 8 replicas, *DCfRSM* mechanism provided around 130 kilobytes of advantage and *Mirrored Solution* provided around 108 kilobytes of advantage compared to *Centralized Solution*. For cluster with 10 replicas, *DCfRSM* mechanism provided around 140 kilobytes of advantage and *Mirrored Solution* provided around 115 kilobytes of advantage compared to *Centralized Solution*.

As presented on Table 5.3, using *DCfRSM* approach also significantly decreased the memory consumption of whole cluster when the number of RSM instances are increased in the cluster. The reason behind this fact is increasing number of replicas means each of the replicas in the cluster will process *less* messages when total of messages processed inside cluster is fixed to *3600* and hence, less memory footprint.

Figure 5.6 also shows the same advantage if *DCfRSM* approach is chosen as primary checkpoint solution in RSM systems in cloud environment. *DCfRSM* mechanism provided around 80 kilobytes of advantage compared to *Mirrored Solution* and around

Table 5.3 : Average KB of memory footprint regarding 4,6,8 and 10 replicas

Distributed	Centralized	Conventional	Mirrored
122,40	86,40	230,40	158,40
110,40	86,40	230,40	133,40
104,40	86,40	230,40	122,40
94,40	86,40	230,40	115,20

382 kilobytes of advantage compared to *Conventional Solution*. For cluster for 6 replicas, *DCfRSM* mechanism provided around 400 kilobytes of advantage compared to *Conventional Solution*. For cluster with 8 replicas, *DCfRSM* provided around 420 kilobytes of advantage compared to *Conventional Solution*. For cluster for 10 replicas, *DCfRSM* mechanism provided around 432 kilobytes of advantage and *Mirrored Solution* provided 384 kilobytes of advantage compared to *Conventional Solution*. Experiments in cloud environment also proves that memory consumption of whole cluster is *decreased* while the number of RSM instances are increased within the scope of *DCfRSM* solution.

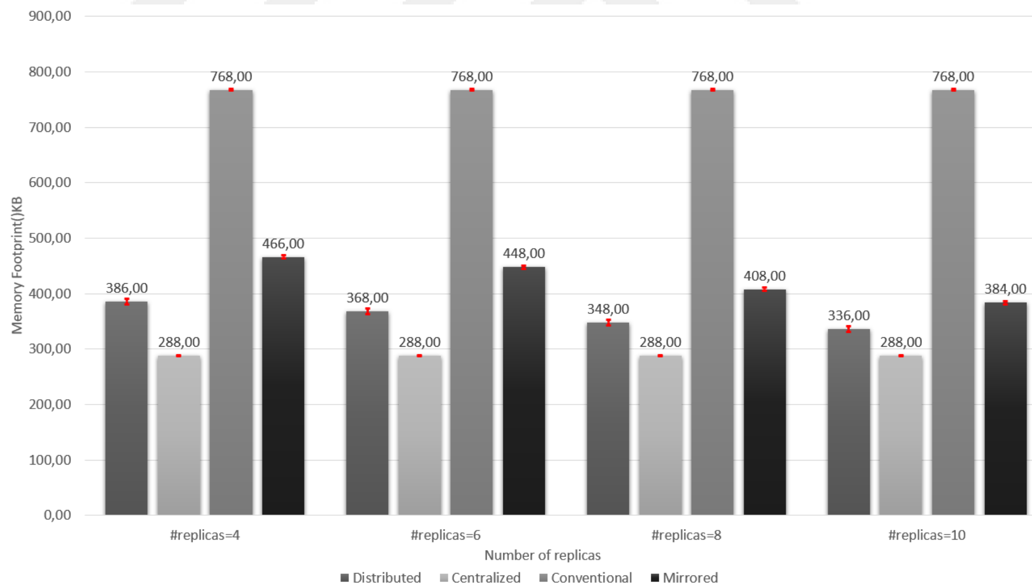

Figure 5.6 : Memory footprint from the experiments in cloud.

Due to Table 5.2, even though there exists an advantage in using *DCfRSM* in terms of memory consumption as per Table 5.3, it seems *DCfRSM* produces high recovery time when it is compared to *Mirrored*, *Centralized* and *Conventional* approaches. Nevertheless, by considering Figure 5.5, it is obvious that all of the solutions implemented spent similar time duration for *JVM Initialization* phase during recovery process while the main difference is caused by the phases *Gather Checkpoints* and

Serialize Checkpoints among implemented solutions. *JVM Initialization* works in pretty straightforward manner and executes standard operations. During this phase, JVM for the execution java programs and java runtime is activated, Java beans are injected, dependency injection mechanisms are executed, state machines are initialized and started. During *Gather Checkpoints* phase which causes significant difference between solutions, all the checkpoint information inside whole cluster has been gathered by recovering/freshly booting RSM instance. For the *Conventional* Solution, checkpoint snapshots has been gathered from only one of RSM instances in the cluster. For the *Centralized* Solution, checkpoint snapshots has been gathered from *LoadBalancer* instance. For the *DCfRSM* and *Mirrored* solutions, several RSM instances will be used for gathering checkpoints. Thus, the time need for this phase has been differentiated solution by solution. *Serialize Checkpoints* phase is only needed for *DCfRSM* and *Mirrored* solutions, on this phase, all the checkpoint snapshots distributed among cluster which is already gathered in previous step has been prepared, serialized and made ready for *Apply Checkpoints* phase. So, during *Apply Checkpoints* phase, all the checkpoints will be applied in the order of execution.

As per Figure 5.7, results from experiments in cloud environment is also compatible with the results generated from docker-compose execution. For all of the replica sets, *DCfRSM* produces less recovery time compared to *Conventional Solution*. In contrast to this statement, *DCfRSM* produces high recovery time once it is compared to *Centralized Solution*. The highest restore duration is produced by *Conventional Solution*. As stated earlier, according to Figure 5.2, once the failing replica is re-joined the cluster again or a brand-new replica is booted from scratch in region *Europe* on instance *Europe-2*, it needs the gather *all the checkpoint snapshots* from a RSM instance in different region -from region *US*- as a whole. While gathering checkpoint snapshots from RSM instance in different region as a whole, communication overhead exist for *Conventional Solution* which leads to having highest restore duration among all the solutions in the cloud environment. This is the only difference experienced between the test experimented on distributed cloud servers and docker-compose execution.

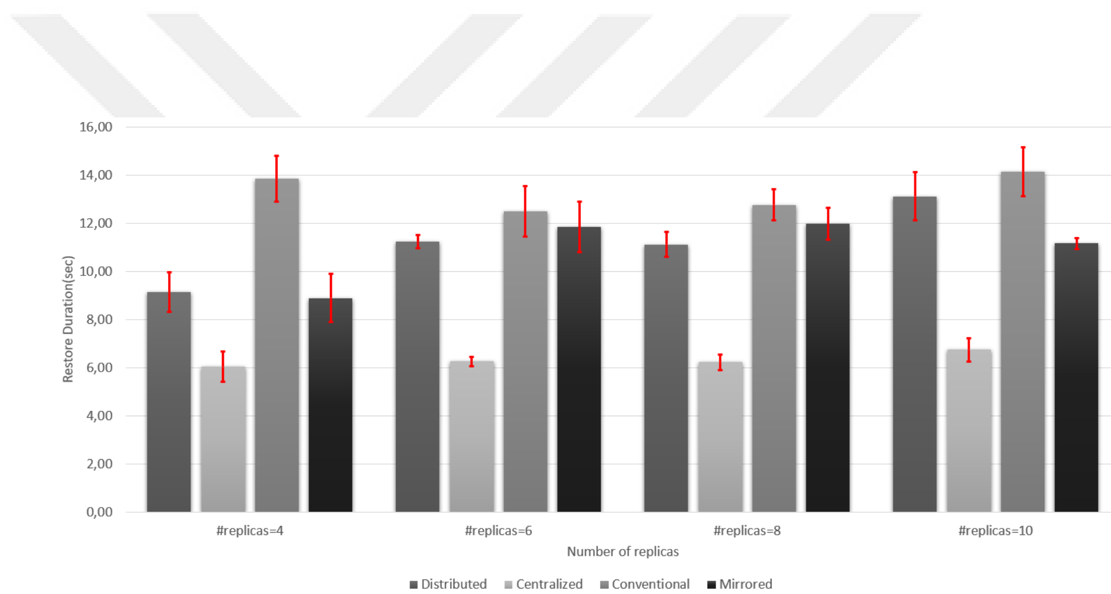

Figure 5.7 : Restore duration from the experiments in cloud.

6. CONCLUSIONS AND FUTURE WORK

In this thesis, a distributed checkpointing algorithm for the replicated state machines is proposed. In order to prove its efficiency, a test bed including few other solutions -conventional, centralized- has been prepared in order to benchmark proposed solution in terms of memory footprint and restore/recovery duration metrics. Afterwards, in order to evaluate and visualize results, graphs has been prepared. Due to results, proposed approach provides less memory usage when it is compared to conventional approach. Besides this advantage, recovery time is increased due to extra communication need for collecting partial histories from replica instances.

It is planned to extend studies with the aim of decreasing recovery time overhead as much as possible in order to provide a better trade-off between *DCfRSM* approach and other approaches -*conventional* and *centralized.*-. The one of the pain points which is needed to be improved is balancing the *communication overhead* among RSM instances during a brand-new RSM instance joining to replica or recovery process of a failed replica. As stated on figure 5.5, another pain point is the time interval while *applying* checkpoints on the new RSM instance or recovering RSM instance. One proposal solution may be *parallelization* of the different phases during recovery process, for instance, *gathering* and *applying* first bunch of checkpoints and then, *gathering* and *applying* second bunch of checkpoints, and so on.

It is also planned that extending this thesis by balancing the amount of checkpoint data stored in each RSM instance. By adding new RSM instances to existing cluster and performing same tests in test bed, less memory footprint is expected by each RSM instance in RSM cluster. As presented by table 5.3, the more RSM instances are deployed on cluster, the less memory footprint needed for each RSM instance as average.

REFERENCES

[1] **Li, C.**, **Porto, D.**, **Clement, A.**, **Gehrke, J.**, **Preguiça, N.M. and Rodrigues, R.** (2012). Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary, *OSDI*.

[2] **Bolosky, William and Bradshaw, Dexter and Haagens, Randolph and Kusters, Norbert and Microsoft, Peng** (2011). Paxos replicated state machines as the basis of a high-performance data store, *NSDI'11: Proceedings of the 8th USENIX conference on Networked systems design and implementation*, 141–154.

[3] **Friedman, R. and Vaysburd, A.** (1997). Fast replicated state machines over partitionable networks, *Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems*, 130–137.

[4] **Cully, B.**, **Lefebvre, G.**, **Meyer, D.**, **Feeley, M.**, **Hutchinson, N. and Warfield, A.** (2008). Remus: High Availability via Asynchronous Virtual Machine Replication. (Best Paper), p.161.

[5] **Heo, Junyoung and Yi, Sangho and Cho, Yookun and Hong, Jiman and Shin, Sung** (2006). Space-efficient page-level incremental checkpointing, *Journal of Information Science and Engineering*, *22*, 237–246.

[6] **Mao, Yanhua and Junqueira, Flavio and Marzullo, K.** (2008). Mencius: Building Efficient Replicated State Machine for WANs., *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 369–384.

[7] **Konczak, J.**, **Santos, N.**, **Zurkowski, T.**, **Wojciechowski, P.T. and Schiper, A.** (2011). JPaxos: State machine replication based on the Paxos protocol.

[8] **B. Ghit and D. H. J. Epema** (2017). Better Safe than Sorry: Grappling with Failues of In-Memory Data Analytics Frameworks, *HPDC*, 105–116.

[9] **Naksinehaboon, Nichamon and Liu, Yudan and Leangsuksun, C. and Nassar, Ruba and Paun, Mihaela and Scott, S.** (2008). Reliability-Aware Approach: An Incremental Checkpoint/Restart Model in HPC Environments, *IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, 783–788.

[10] **Slember, J. and Narasimhan, P.** (2006). Static analysis meets distributed fault-tolerance: enabling state-machine replication with nondeterminism, 2–2.

[11] **Du, J.**, **Sciascia, D.**, **Elnikety, S.**, **Zwaenepoel, W. and Pedone, F.** (2014). Clock-RSM: Low-Latency Inter-datacenter State Machine Replication Using Loosely Synchronized Physical Clocks, pp.343–354.

[12] **Sancho, J.C. and Petrini, Fabrizio and Johnson, G. and Frachtenberg, Eitan** (2004). On the feasibility of incremental checkpointing for scientific computing, *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2004 (Abstracts and CD-ROM)*, *18*, 58–.

[13] **Gioiosa, R. and Sancho, J.C. and Jiang, S. and Petrini, Fabrizio** (2005). Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers, *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 9– 9.

[14] **Rushby, J.M.** (1996). Reconfiguration and transient recovery in state machine architectures, *Proceedings of Annual Symposium on Fault Tolerant Computing*, 6–15.

[15] **Schneider, F. and Zhou, L.** (2010). Implementing Trustworthy Services Using Replicated State Machines, volume 3, pp.151–167.

[16] **Wester, Benjamin and Cowling, James and Nightingale, Edmund B. and Chen, Peter M. and Flinn, Jason and Liskov,Barbara** (2009). Tolerating latency in replicated state machines through client speculation, *6th USENIX Symposium on Networked Systems Design amp; Implementation (NSDI)*.

[17] **Kuwahara, T.**, **Kuroda, T.**, **Nakanoya, M.**, **Yakuwa, Y.**, **Sato, Y. and Matsunaga, Y.** (2019). Automated Planning of System Rollback in Declarative IT System Update, *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 428–434.

[18] **Mendizabal, O.M.** (2016). Fast recovery in parallel state machine replication.

[19] **Konczak, J.**, **Wojciechowski, P.**, **Santos, N.**, **Zurkowski, T. and Schiper, A.** (2019). Recovery Algorithms for Paxos-based State Machine Replication, *IEEE Transactions on Dependable and Secure Computing*, *PP*, 1–1.

[20] **Zhao, Wenbing** (2016). Performance optimization for state machine replication based on application semantics: A review, *Journal of Systems and Software*, *112*, 96–109.

[21] **Fred B. Schneider** (1990). Implementing fault-tolerant services using the state machine approach: A tutorial, *ACM Computing Surveys*, *22*, 299–319.

[22] **G, A.N.** (2016). Design of Fault Tolerant State Machine for a Configurable RISC Processor, volume 5, pp.127–132.

APPENDICES

APPENDIX A : RSM implementation source code
APPENDIX B : UML diagrams
APPENDIX C : LoadBalancer implementation source code

APPENDIX A

CheckpointDbobject.java

```java
package tr.edu.itu.bbf.cloudcore.distributed.persist;

/* DTO = Data Transfer Object */

import org.springframework.data.annotation.Id;

import java.util.UUID;

public class CheckpointObject {

    @Id
    public String timestamp;
    private String context;
    private UUID uuid;
    private String processedEvent;
    private String sourceState;
    private String targetState;
    private Integer eventNumber;


    public CheckpointObject(String timestamp, UUID uuid,
        String sourceState, String processedEvent,
        String targetState, String context,
        Integer eventNumber) {
        this.timestamp = timestamp;
        this.uuid = uuid;
        this.sourceState = sourceState;
        this.processedEvent=processedEvent;
        this.targetState = targetState;
        this.context = context;
        this.eventNumber = eventNumber;
    }


    public String getProcessedEvent(){
        return this.processedEvent;
    }
    public UUID getUuid(){
        return this.uuid;
        }
    public String getSourceState(){
        return this.sourceState;
        }
    public String getTargetState(){
        return this.targetState;
        }
    public String getContext(){
        return this.context;
        }
    public Integer getEventNumber(){
        return this.eventNumber;
        }

}
```

CheckpointService.java

```java
package tr.edu.itu.bbf.cloudcore.distributed.service;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.Message;
import org.springframework.stereotype.Service;
import tr.edu.itu.bbf.cloudcore.distributed.persist.CheckpointDbObject;

import javax.annotation.PostConstruct;
import java.util.Calendar;
import java.util.List;
import java.util.UUID;
import java.util.ArrayList;
import itu.distributed.ObjectSizeFetcher;

@Service
public class RouterService {

    static final Logger logger =
        LoggerFactory.getLogger(RouterService.class);

    @Autowired
    private CheckpointDbObjectHandler dbObjectHandler;

    private ArrayList<CheckpointDbObject> ckptList;

    @PostConstruct
    public void init() {
        ckptList = new ArrayList<CheckpointDbObject>();
    }

    public void storeCKPTInMemory(Message<String> ckptMessage){
        /* Get state machine UUID */
        Object O_UUID =
            ckptMessage.getHeaders().
            get("machineId");
        UUID uuid =
            UUID.fromString(O_UUID.toString());
        /* Get processed event */
        String processedEvent =
            ckptMessage.getHeaders().
            get("processedEvent").toString();
        /* Get source and target states */
        String sourceState =
            ckptMessage.getHeaders().
            get("source").toString();
        String targetState =
            ckptMessage.getHeaders().
            get("target").toString();
        /* Get SMOC context */
        String context =
            ckptMessage.getHeaders().
            get("context").toString();
        /* Store in memory */
        CheckpointDbObject dbObject =
            new CheckpointDbObject(
                getTimeStamp(), uuid, sourceState,
                processedEvent, targetState,
                context,eventNumber);
        ckptList.add(dbObject);
        long currMemoryFootprint =
            (ckptList.size()) *
            (ObjectSizeFetcher.getObjectSize(dbObject));
        logger.info("Checkpoint␣Size␣=␣{}", currMemoryFootprint);
    }
```

```
    public String getTimeStamp(){
        Calendar now = Calendar.getInstance();
        int year = now.get(Calendar.YEAR);
        int month = now.get(Calendar.MONTH) + 1;
        int day = now.get(Calendar.DAY_OF_MONTH);
        int hour = now.get(Calendar.HOUR_OF_DAY);
        int minute = now.get(Calendar.MINUTE);
        int second = now.get(Calendar.SECOND);
        int ms = now.get(Calendar.MILLISECOND);


        String ts =
            year + "." + month + "." +
            day + "_" + hour + "." +
            minute + "." + second + "." + ms;
        return ts;
    }

}
```

StateMachineConfiguration.java

```
package tr.edu.itu.bbf.cloudcore.distributed.config;


import org.jetbrains.annotations.NotNull;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.statemachine.StateContext;
import org.springframework.statemachine.action.Action;
import org.springframework.statemachine.config.*;
import org.springframework.statemachine.config.builders.*;
import tr.edu.itu.bbf.cloudcore.distributed.entity.Events;
import tr.edu.itu.bbf.cloudcore.distributed.entity.States;
import tr.edu.itu.bbf.cloudcore.distributed.listener.*;

import javax.annotation.PostConstruct;
import java.util.Map;
import java.util.UUID;
import java.util.concurrent.TimeUnit;

@Configuration
@EnableStateMachineFactory(name = "factory_without_ZK")
public class StateMachineConfig_WithoutZK
    extends EnumStateMachineConfigurerAdapter<States, Events> {


    private Logger logger = LoggerFactory.getLogger(getClass());

    /** Default constructor **/
    public void StateMachineConfig_WithoutZK(){}

    @PostConstruct
    public void init() {}

    @Override
    public void
        configure(StateMachineStateConfigurer<States, Events> states)
            throws Exception {
        states.withStates()
                .initial
                (States.UNPAID, _initializationAction())
                .stateEntry
                (States.WAITING_FOR_RECEIVE, _entryActionForWaiting)
                .stateExit
                (States.WAITING_FOR_RECEIVE, _exitActionForWaiting)
                .stateEntry
                (States.DONE, _entryActionForDone())
                .stateExit
                (States.DONE, _exitActionForDone());
    }
       .machineId("machine-without-zk");
    }
```

```java
@Override
public void
    configure(StateMachineTransitionConfigurer<States, Events> transitions)
        throws Exception {
    /** Defines "EXTERNAL" type of transitions **/
    transitions
            .withExternal()
            .source(States.UNPAID)
            .target(States.WAITING_FOR_RECEIVE)
            .event(Events.PAY)
            .action(_transitionAction())
            .and()
            .withExternal()
            .source(States.WAITING_FOR_RECEIVE)
            .target(States.DONE)
            .event(Events.RECEIVE)
            .action(_transitionAction())
            .and()
            .withExternal()
            .source(States.DONE).target(States.UNPAID)
            .event(Events.STARTFROMSCRATCH)
            .action(_transitionAction());
}

@Bean
public Action<States, Events> _entryActionForWaiting() {
    return new Action<States, Events>() {
    @Override
    public void execute(StateContext<States, Events> context){
            Integer localVar =
                context.getExtendedState().
                get("localVarForWaiting", Integer.class);
            localVar = localVar + 2;
            context.getExtendedState().
                getVariables().
                put("localVarForWaiting", localVar);
        }
    };
}

@Bean
public Action<States, Events> _exitActionForWaiting() {
    return new Action<States, Events>() {
    @Override
    public void execute(StateContext<States, Events> context){
            Integer localVar =
                context.getExtendedState().
                get("localVarForWaiting", Integer.class);
        }
    };
}
```

```
    @Bean
    public Action<States, Events> _entryActionForDone() {
        return new Action<States, Events>() {
        @Override
        public void execute(StateContext<States, Events> context){
                Integer localVar =
                    context.getExtendedState().
                        get("localVarForDone", Integer.class);
                localVar = localVar + 5;
                context.getExtendedState().
                        getVariables().
                        put("localVarForDone", localVar);
            }
        };
    }

    @Bean
    public Action<States, Events> _exitActionForDone() {
        return new Action<States, Events>() {
        @Override
        public void execute(StateContext<States, Events> context) {
                Integer localVar =
                    context.getExtendedState().
                        get("localVarForDone", Integer.class);
            }
        };
    }

    @Bean
    public Action<States, Events> _initializationAction() {
        return new Action<States, Events>() {
        @Override
        public void execute(StateContext<States, Events> context){
                /** Define extended state variable
                **  as common variable used
                **  inside transition actions **/
                context.getExtendedState().
                    getVariables().put("common", 0);
                /** Define extended state variable
                **  as private/local variable
                **  used inside state actions **/
                context.getExtendedState().
                    getVariables().put("localVarForWaiting",10);
                context.getExtendedState().
                    getVariables().put("localVarForDone",50);
            }
        };
    }

    public void sleepForAWhile(Long sleepTime){
        try {
            TimeUnit.MILLISECONDS.sleep(sleepTime);
        } catch (InterruptedException ex) {}

    }

}
```

```
    @Bean
    public Action<States, Events> _transitionAction() {
        return new Action<States, Events>() {
        @Override
        public void execute(StateContext<States, Events> context){
                Map<Object, Object> variables =
                    context.getExtendedState().
                        getVariables();
                Integer commonVar =
                    context.getExtendedState().
                        get("common", Integer.class);

                if (commonVar == 0) {
                    variables.put("common", 1);
                    sleepForAWhile(longSleep);
                } else if (commonVar == 1) {
                    variables.put("common", 2);
                    sleepForAWhile(longSleep);
                } else if (commonVar == 2) {
                    variables.put("common", 0);
                    sleepForAWhile(longSleep);
                }

                try {
                    PerformCheckpoint(context);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        };
    }

}
```

APPENDIX B

UML diagrams

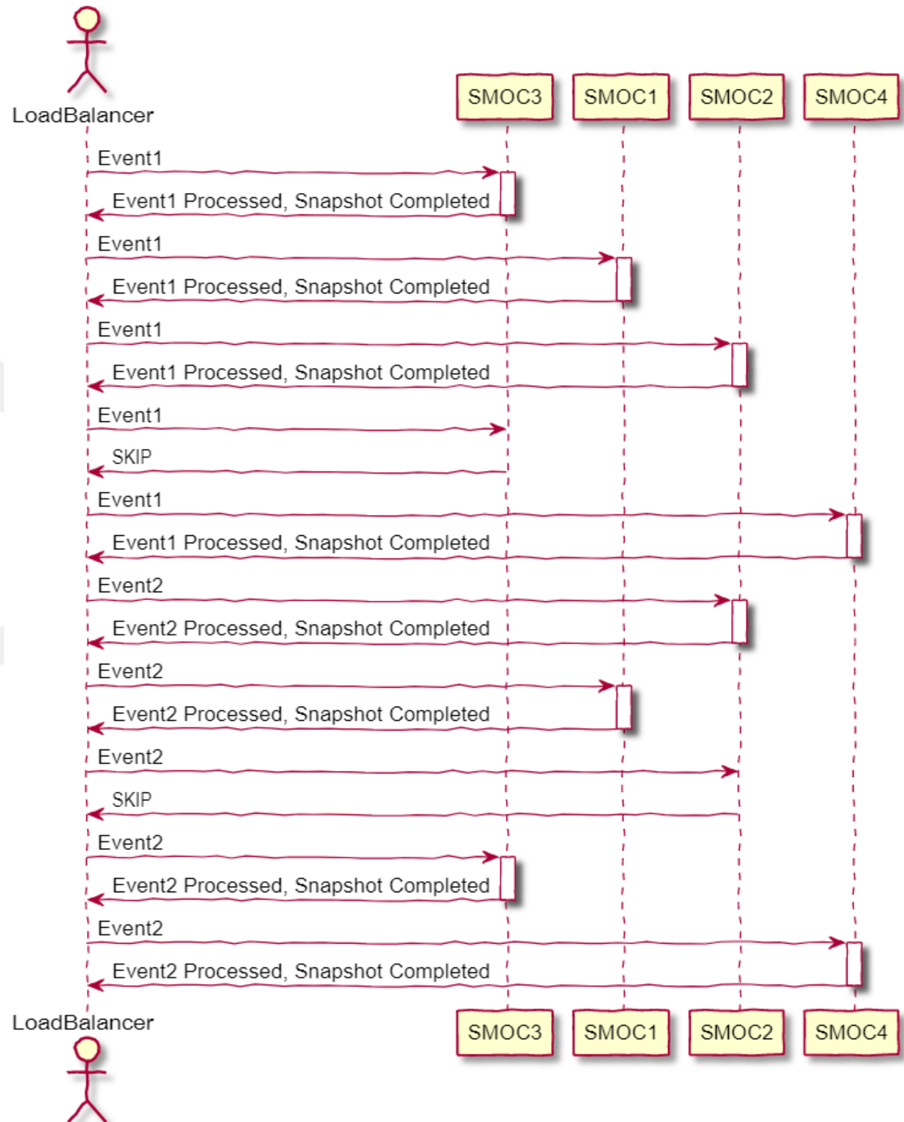

Figure B.1 : UML diagram for the conventional solution.

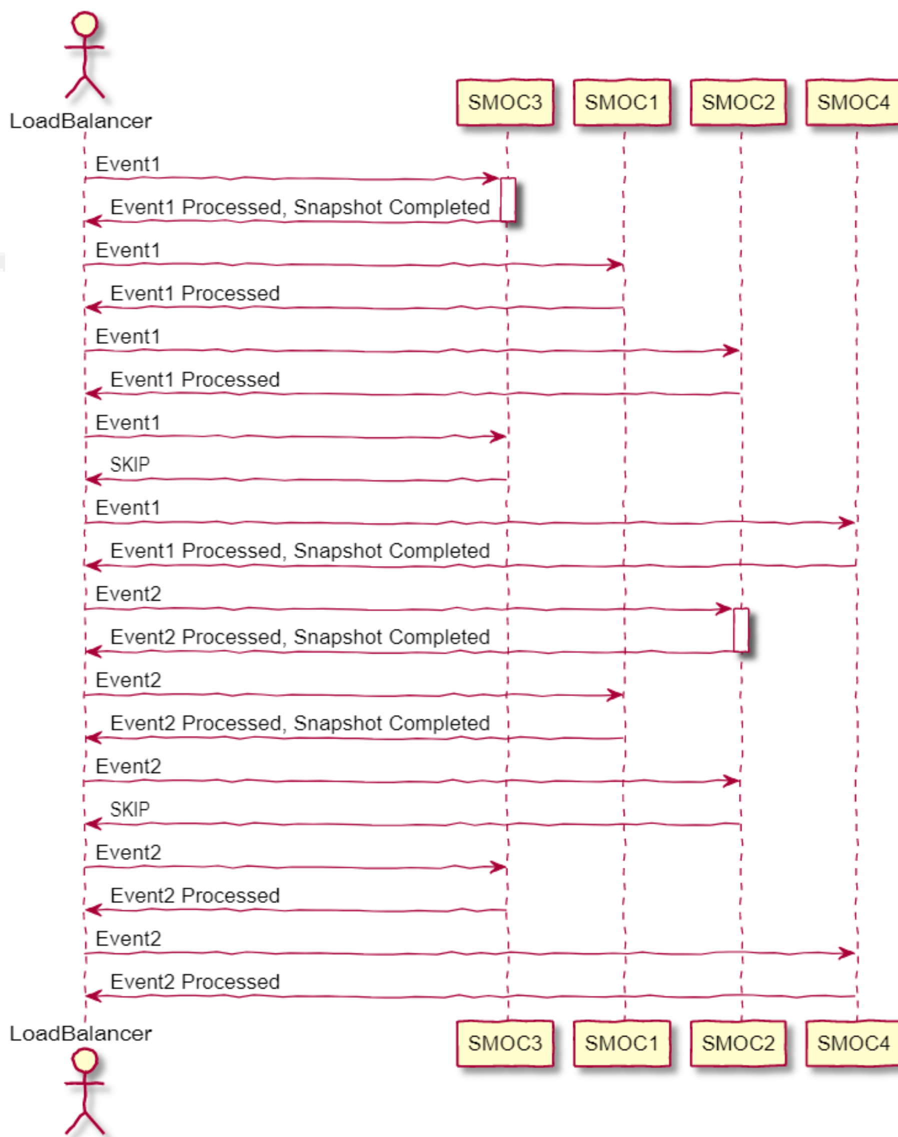

Figure B.2 : UML diagram for the DCfRSM.

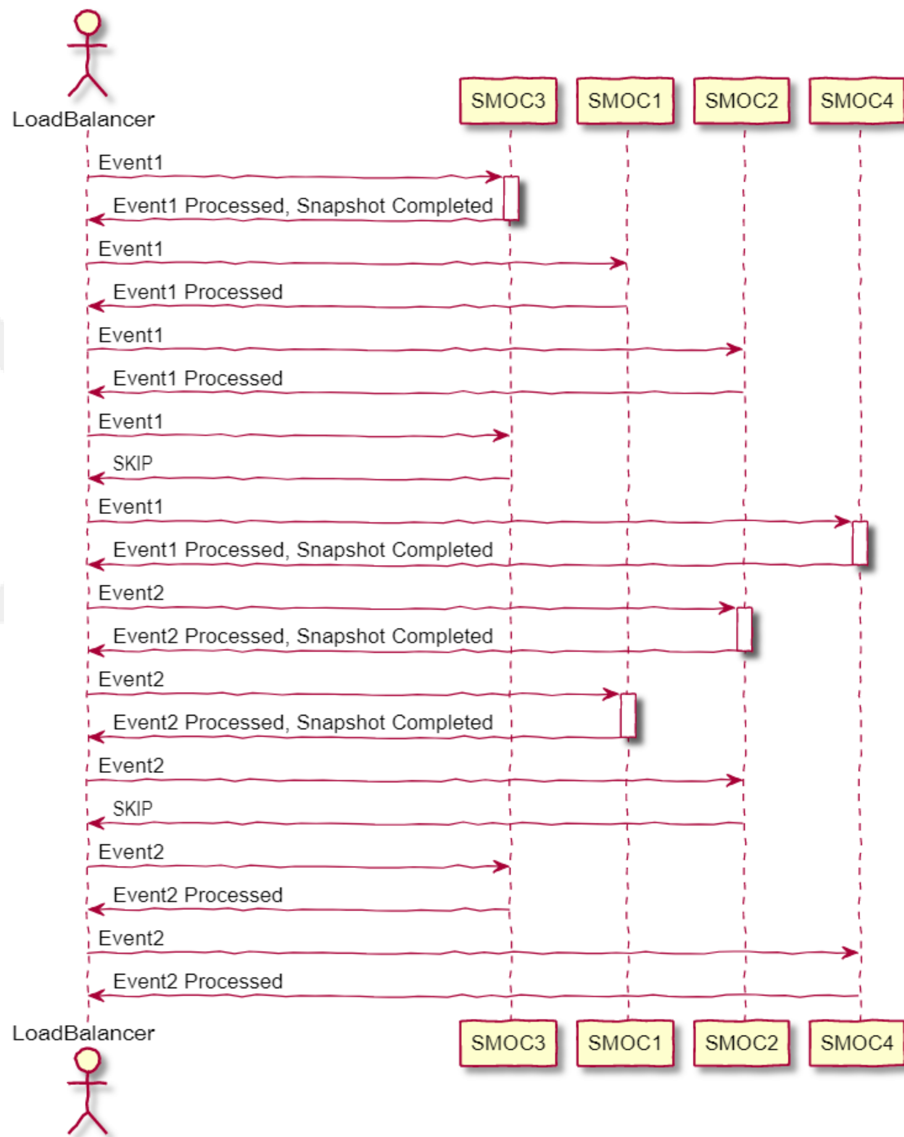

Figure B.3 : UML diagram for the mirrored solution.

APPENDIX C

LoadBalancer.java

```java
package tr.edu.itu.bbf.cloudcore.distributed;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.*;
import org.springframework.context.annotation.*;
import org.springframework.core.env.Environment;
import tr.edu.itu.bbf.cloudcore.distributed.ipc.EventSender;
import tr.edu.itu.bbf.cloudcore.distributed.service.InMemoryStore;

import java.util.*;


@ComponentScan(basePackages = {"tr.edu.itu.bbf.cloudcore.*"})
@SpringBootApplication
public class Application implements CommandLineRunner {

    @Autowired
    private EventSender sender;

    @Autowired
    private InMemoryStore inMemoryStore;

    @Autowired
    private Environment environment;

    private String workingMode;

    private enum Events{Pay, Receive, StartFromScratch}

    private enum Hosts
        {SMOC1, SMOC2, SMOC3, SMOC4, SMOC5,
        SMOC6, SMOC7, SMOC8, SMOC9, SMOC10,
        SMOC11, SMOC12, SMOC13, SMOC14, SMOC15}

    private Dictionary peerGroup;

    private Integer eventNumber;
```

```java
@Override
public void run(String... args) throws Exception {
    this.peerGroup = new Hashtable();

    this.eventNumber = 1;
    Integer event = 0;
    Integer eventIndex = -1;

    // iterate over enums using for loop
        switch (workingMode){
            case "WithoutExtraCheckpoint":
                while(event < numberOfEvents) {
                    eventIndex = event % 3; // since we have 3
                    Hosts randomHost =
                        Hosts.values()
                        [new Random().nextInt(numberOfReplicas)];
                    Events eventToBeProcessed =
                        Events.values()[eventIndex];
                    sendEventToEnsemble
                    (
                    eventToBeProcessed.toString(),
                    randomHost.toString(),
                    numberOfReplicas, false, false);
                    event = event + 1;
                }
                break;

                case "WithExtraCheckpoint":
                while(event < numberOfEvents) {
                    // since we have 3 event transitions
                    eventIndex = event % 3;
                    Events eventToBeProcessed =
                        Events.values()[eventIndex];
                    Hosts firstHost =
                        Hosts.values()[numberOfReplicas - 1];
                    sendEventToEnsemble
                    (
                    eventToBeProcessed.toString(),
                    firstHost.toString(),
                    numberOfReplicas, true,true);
                    event = event + 1;
                }
                break;
```

```
                case "PartialCheckpoint":
                    while(event < numberOfEvents) {
                        // since we have 3 event transitions
                        eventIndex = event % 3;
                        Events eventToBeProcessed =
                            Events.values()[eventIndex];
                        Hosts randomHost =
                            Hosts.values()[
                            // start from random smoc
                            new Random().nextInt(numberOfReplicas)];
                        sendEventToEnsemble
                        (
                        eventToBeProcessed.toString(),
                        randomHost.toString(),
                        numberOfReplicas, false, true);
                        event = event + 1;
                    }
                    break;
            }

    }


    public String whoIsMyPair(String host){
        /* n <- smoc<n>*/
        Integer smocNumber = Integer.parseInt(host.substring(4));
        Integer peerGroup = -1 ;
        String peer = "";
        if (smocNumber % 2 == 0)
        { /* EVEN */
            /* Peer Group = 10 <- smoc20 */
            peerGroup = smocNumber/2;
            peer = (
            (ArrayList<String>)this.peerGroup.
            get(peerGroup)).get(0);
        }
        else
        { /* ODD */
            /* Peer Group = 10 <- smoc19 */
            peerGroup = (smocNumber+1)/2;
            peer = (
            (ArrayList<String>) this.peerGroup.
            get(peerGroup)).get(1);
        }
        return peer;
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

CURRICULUM VITAE

Name Surname: Niyazi Ozdinc Celikel

Place and Date of Birth: Çanakkale - 17 January 1991

E-Mail: ozdinc.celikel@gmail.com

EDUCATION:

- **B.Sc.:** 2014, Istanbul Technical University, Faculty of Computer and Informatics Engineering, Computer Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2014 - 2018 - Arçelik - System Engineer
- 2018 - ... - Siemens - Software Developer

PRESENTATIONS ON THE THESIS

Celikel N. Ozdinc, Ovatman T., 2020: A Distributed Checkpoint Mechanism for Replicated State Machines. *Accepted and presented in CLOSER 2020, 10th International Conference on Cloud Computing and Services Science (CLOSER)*, 7-9 May. 2020