Scalable Computing: Practice and Experience, ISSN 1895-1767, http://www.scpe.org
© 2021 SCPE. Volume 22, Issues 1, pp. 67–79, DOI 10.12694:/scpe.v22i1.1840

DISTRIBUTED APPLICATION CHECKPOINTING FOR REPLICATED STATE MACHINES[*]

NIYAZI ÖZDINÇ ÇELIKEL[†] AND TOLGA OVATMAN[‡]

Abstract. Application checkpointing is a widely used recovery mechanism that consists of saving an application's state periodically to be used in case of a failure. In this study we investigate the utilisation of distributed checkpointing for replicated state machines. Conventionally, for replicated state machines, checkpointing information is stored in a replicated way in each of the replicas or separately in a single instance. Applying distributed checkpointing provides a means to adjust the level of fault tolerance of the checkpointing approach by giving away from recovery time. We use a local cluster and cloud environment to examine the effects of distributed checkpointing in a simple state machine example and compare the results with conventional approaches. As expected, distributed checkpointing gains from memory consumption and utilise different levels of fault tolerance while performing worse in terms of recovery time.

Key words: Application Checkpointing, Replicated State Machines, Cloud Computing

AMS subject classifications. 68M14, 68W15

1. Introduction.

During the passing few years, serverless computing has become more widespread among the cloud service providers. Very broadly, this term refers to isolating almost every layer of the software development stack from service developer by providing a service modelling medium such as a state machine. By using this service definition model, developer might model and execute simple services without worrying about the configuration of software stack layers.

From a cloud provider's perspective, using replicated state machine (RSM) approach for fault tolerance is a favourable alternative [1] [2] since it is a widely-known and implemented approach among software developers, there even exists many frameworks for back-end programming such as Spring State Machines[1]. RSMs simply execute replicas of a state machine to handle requests in a distributed way. During running time, each replica handles different requests and executes them as if they are being orderly processed by a main state machine.

An example of state machine replication can be seen in Fig. 1.1, where a master state machine on top is replicated over three replicas. State machines transit between defined states such as A, B and C with incoming events such as E1, E2 and E3. Using a master replica (or state machine) is dependent on the context of usage. When no master is used, replicas are expected to eventually be orchestrated to reflect a single logical state machine. The replicas can be deployed in proximate locations as well as in geographically distinct locations [3] that may affect the performance of orchestration among the replicas.

One of the important aspects in deploying RSMs is fault recovery. Replicas may periodically save system state, known as checkpoints, to recover to a past state in the presence of a system failure. Checkpointing is also utilised for the cases where a new replica is introduced to the system to update the replica's state to the current state of the RSM. For RSMs, using different checkpointing approaches might have different characteristics in terms of non-functional properties of the system. For instance, if each replica keeps full restore information specific to the replica, redundant replicated checkpointing information would emerge since all the replicas eventually go through the same execution path at run-time. On the other hand, keeping a single checkpointing replica would result in a single point of failure for checkpointing operation.

[*]This work is supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under grant id 118E887.

[†]Istanbul Technical University Department of Computer Engineering Istanbul, Turkey (celikelni@itu.edu.tr).

[‡]Istanbul Technical University Department of Computer Engineering Istanbul, Turkey (ovatman@itu.edu.tr) ORCID-id:0000-0001-5918-3145. Corresponding author.

[1]https://projects.spring.io/spring-statemachine/

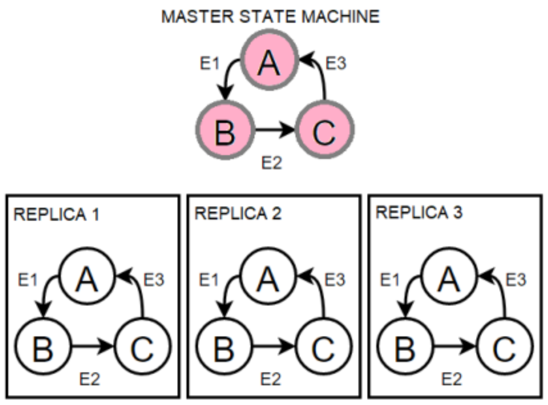

Fig. 1.1: State machine replication

In this study, we utilise characteristics specific to state machines to introduce a distributed checkpointing approach for RSMs (DCfRSM) that simply distributes the checkpointing operation and checkpoint information among replicas. Our approach makes it possible to utilise different levels of replication of checkpointing information to leverage recovery overhead and fault tolerance. We use a simple state machine instance, implemented using spring state machines to demonstrate and evaluate our approach. We use different number of replicas running in a local cluster and in amazon web services separately to measure memory consumption for each replica and recovery time for a booting state machine replica. We compared our approach with two different approaches, namely conventional checkpointing and centralised checkpointing, to evaluate the advantages in using distributed checkpointing for RSMs. Results from these experiments show that DCfRSM provided advantage in terms of memory consumption compared to centralised conventional approaches. On the other hand, DCfRSM produces a high recovery time when it is compared to centralised and conventional approaches because of the extra communication overhead needed for collecting partial histories from different replicas inside a cluster. However, we believe this overhead is the cost of obtaining a higher level of fault tolerance especially with respect to centralised checkpointing. This study expands our earlier preliminary study [4] by providing implementation details of the DCfRSM approach and results from experiments on a real cloud environment.

The rest of the paper is organised as follows: In Sect. 2 we review related literature. In Sect. 3 we explain the DCfRSM approach in more detail. Experimental architecture, implemented approaches for benchmarking purposes and simulation environment are introduced in Sect. 4. Section 5 presents the results of the experiments and the paper is concluded in Sect. 6.

2. Related Work.

There exists numerous studies to optimise performance and recovery costs of checkpointing approaches in distributed computing. In this study, we study on application level checkpointing which is applied in a more software-agnostic way by relying on operating on memory as a whole; a comparative discussion between system and application level checkpointing can be found in [5]. The work depicted in [6] states the necessity for replicated state machines to guarantee that majority of replicas inside a cluster can communicate with each other and be prone to node failures. In case of failures on physical machines, in order to minimise checkpointing costs, [7] proposes a novel replication technique with the aim of decreasing recovery costs while [8] proposes a new approach for reducing storage costs.

Due to the review by [9], several layers of fault-tolerance may be defined, such as optimistic fault-tolerance and conservative fault-tolerance mechanisms. This study 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, is also one of the features proposed by the DCfRSM approach.

The idea behind using replicated state machines in order to model distributed checkpointing approach is already stated by [2] and [10]. 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 study. In this aspect, another interesting study is abortable state machine replication approach [11], implemented as an extension for Zyzzyva [12] where authors provide a byzantine fault tolerance mechanism to support interruption of execution in replicated state machines.

Moreover, the study in [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], we also experiment on cloud systems to be able to examine the advantage of DCfRSM approach on geographically distributed and replicated state machines.Thesis study in [13], also represents an efficient logging mechanism along with an efficient checkpointing model, which is executed in a parallel and distributed manner by executing concurrent commands. By using this approach, not only recovery process is parallelised but also checkpointing is persisted concurrently in all replicas.

Following geographical distribution, a number of studies has also been publishing the utilisation of checkpointing in cloud environments and in state machines running on cloud environments. Providing checkpointing, as a cloud service has been proposed in an earlier study, where authors have used existing software packages to implement checkpointing on cloud environments [14]. A later study examined checkpointing in edge cloud scenarios and provided algorithms to improve persistence and recovery server selection processes [15]. Having a similar domain with edge domain, a past study uses state machine models of internet of things devices to select optimal points for checkpointing and try to reduce energy overhead of checkpointing process [16].

Another area of literature, regarding checkpointing in the cloud, consists recovery processes of distributed running tasks. A recent study proposes a system in this aspects and evaluates over energy consumption, service level agreement violations, recovery time of tasks and failure rates [17]. A very recent study also reports storage checkpoint recovery times for bag-of-tasks jobs over Amazon Web Services [18]. Even though not being in cloud domain, there has been past research on modelling tasks as state machines and using state machine properties to schedule checkpointing process to optimise restoration time [19].

Reduction of communication between replicas is not the primary concern of our study but there are studies in literature focusing on networking aspects. For instance, the study in [20] proposes a high-performance replicated state machine checkpoint and recovery approach inspired by Paxos consensus protocol. There are also other studies that utilise Paxos such as [21], where authors propose an efficient implementation for snapshots and recovering current state of the state machine.

In order to minimise overall checkpoint overhead, the study in [22] proposes to checkpoint only straggling tasks in order to minimise the number of checkpoints. Within the scope of our study, instead of persisting checkpoints after only certain tasks, as suggested by [22], we have chosen to persist checkpoints to be triggered just after every task execution inside state machines.

With the aim of reducing the checkpoint data size, the study depicted in [23] propose a novel checkpointing mechanism by modelling a decision algorithm in order to reveal the and persist dirty pages that are modified since last checkpoint time. We employ the time ticks and execution history elements in incremental checkpointing approach introduced by [23] and store events occurred within predesignated time ticks for a predesignated execution steps, instead of forcing all the replicas to checkpoint all the modifications performed on the internal states so far. Another study in [24] presents concurrent replication technique for the replicated state machines and compensate non-determinism with the help of static analysis. In our study we reduce the checkpointing storage costs by trying to eliminate redundant checkpointing information from replicas.

Another important aspect of implementing a checkpointing approach is the system level which the designated approach is going to operate on; such as in user level [25] or kernel level [26]. The approach introduced in this paper operates as user level. In the study depicted by [25], states that the user level checkpointing is performed explicitly by external applications and hence, user-level application is unaware whether it is check-pointed or not.On the other hand, the study in [26] 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 operating system's signal mechanism can easily be used for checkpointing formulations.

Although considerable amount of work has been performed on memory checkpointing, very few recent studies exists that provides an approach utilising state machines. A very recent example to such a study uses checkpointing in persisting distributed legacy in memory software by the introduction of a persistent memory based tool [27]. Another recent study in non volatile memory systems uses differential checkpointing to leverage energy efficiency [28]. Even though state machines are not explicitly used in this study, a recent application of differential checkpointing is presented.

Checkpointing in in-memory processing has been focus of recent studies; an example to such a study is the idea of applying probabilistic checkpointing on the domain of stream processing where authors present a periodic multi-level checkpointing approach and evaluated their approach by experimenting on Apache Flink [29]. An earlier study proposes asynchronous checkpointing approach to be used in in-memory database systems by defining virtual consistency points in application run and apply checkpointing regarding those points [30]. Frequency of checkpointing, lately has drawn some attention as well; a recent study explores how recomputing some data values instead of recovering a persistent copy may decrease checkpointing frequency and provide energy efficiency [31].

Besides checkpointing approaches, there has also been interest in recovery mechanisms with respect to state machine execution context. Due to the study in [32], there are various industry-standard tools which adopts recovery approach in the context of state machines in different aspects. An example to such an approach is "declarative system update", that works by defining the desired state of system and applying necessary modifications to current state in order to achieve the desired state by using RSMs in different context. In addition to this study, the study in [13] increases the performance of the recovery of failed replicas by parallelising the checkpointing operation. Parallelisation, in this study, is achieved by execution of concurrent commands under coordinated and uncoordinated modes of execution. This approach provides a chance for achieving consistency for both faulty and regular(non-faulty) replicas. The study depicted in [33], proposes three novel recovery approaches that produce less overhead during restoration in faulty replicas. Our proposed approach is also inspired from this study in reducing the amount of overhead produced by replicas by reducing the amount of extra processing related to checkpointing.

Efficient checkpointing and recovery mechanisms in the context of replicated state machines has other application areas as well. An example to such an application area can be found in [34], where efficient recovery execution is implemented in the presence of arbitrary faults. The study depicted in [35], proposes to use divide-and-conquer approach for the fault-tolerant replicated state machine cluster systems. According to the study in [36], a decision system inside state machine cluster may predict the executing process being CPU-bound or I/O-bound. According to this decision, subsequent modifications are speculatively executed and used in checkpointing. 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. This approach is stated as beneficial if the time interval of checkpointing is less than the time interval of performing operation which generates the expected result.

3. Distributed Checkpointing for Replicated State Machines (DCfRSM).

Distributed checkpointing approach employs deploying and serialising request history handled by an RSM into many pieces during persisting checkpoints. This way, each RSM instance may store a specific piece of history instead of full execution history. During a recovery, whole history is going to be gathered from the components of the system, which also means, logical master history will be shared between all the active state machine replicas.

Definition 3.1 (State Machine). *A state machine is composed of a triplet where S is a set consisting the states in the system, E is the set of events and F is a transition function that represent transitions between the states, each triggered by an event.*

$$\begin{aligned}
&M = \{S, E, F\} \\
&S = \{s_0, s_1, \ldots\} \\
&E = \{e_0, e_1, \ldots\} \\
&F \subset S \times E \times S
\end{aligned}$$

To explain the distributed checkpointing approach in more detail we employ a labelled state transition model where a state machine is defined with a triplet $M = \{S, E, F\}$ such as in Definition 1. In this model S represents the set of states in the transition system, E corresponds to the set of labels used to label the transitions between the states and F is the transition function between the states that defines a deterministic system.

More precisely, each and every RSM instances includes some states stated as $s_i \in S$, some labels corresponding to events $e_i \in E$ triggering transitions between state machine states such as $s_i \overset{e_k}{\rightarrow} s_j$. The transition function F is defined over $S \times E \rightarrow S$. For instance for the master state machine in Fig. 1.1, $S = \{A, B, C\}$ where $E = \{E1, E2, E3\}$ and $F = \{(A, E1, B), (B, E2, C), (C, E3, A)\}$.

We may use the generic machine definition in presented in Definition 1 to demonstrate the distributed checkpointing process. Whole execution history for the RSM instance can be illustrated as in definition in Eq. 3.1. By using this equation, it is possible to state the history begins with a designated state, execution of state machine continues by events that trigger the machine to transit between the states.

$$H = (s_i, e_i, s_j, e_j, s_k, e_k, s_m)\dots \tag{3.1}$$

An execution history instance contains some number of events that results in the machine to transit between states in an orderly manner. In order to represent this order of events we may use superscripts to annotate our history definitions such as in Eq. 3.2[2]. Here, we omit the subscripts that distinguish between the specific events/states for simplicity. In case of a replicated state machine, eventually, each replica of the master state machine is supposed to execute the same order of events. Hence, in a synchronisation agnostic manner, we may distribute the responsibility of saving specific parts of history to specific machines.

$$\begin{aligned}
H^{[0-59]} &= (s^0, e^0, s^1, e^1, s^2, e^2, \dots, s^{59}, e^{59}, s^{60}) \\
H_0^{[0-19]} &= (s^0, e^0, s^1, e^1, \dots, s^{19}, e^{19}, s^{20}) \\
H_1^{[20-39]} &= (s^{20}, e^{20}, s^{21}, e^{21}, \dots, e^{39}, s^{40}) \\
H_2^{[40-59]} &= (s^{40}, e^{40}, e^{41}, e^{42}, \dots, e^{59}, s^{60})
\end{aligned} \tag{3.2}$$

A very straightforward example would be the one in Eq. 3.2, where the history is divided into three equally length parts. A division like in Eq. 3.2 might be accomplished in the presence of three replicas which has executed 60 events so far. Each replica saves a specific portion of history which might be represented as H_i^τ where i represents the replica id and τ represents the time interval which replica needs to save the history for the checkpointing purpose. Once the τ is parameter is determined for the overall system, a specific replica might simply perform history saving decision by a simple arithmetic operation. For instance for a three replica system where τ is designated as 20, the replica with id 0 should begin saving history for 20 events every time the modulus of the event number divided by τ equals its own id. Equation 3.3 formalises this calculation by representing replica id by r_{id}[3], event number by e_i and number of replicas by $|R|$ where R corresponds to the replica set.

$$r_{id} == ((e_i \texttt{ div } \tau) \texttt{ mod } |R|) \tag{3.3}$$

In this decision process, an important aspect would be the necessity to broadcast and synchronise whenever a new replica joins the replica set or a present replica leaves the replica set since those situations change the specific points in history where a replica starts saving history for checkpoint. Another important aspect is distributing the history portions in the aforementioned way works correctly for the case when a new replica joins the system but in case of a failure, the specific portion of the history saved by the failing replica becomes lost. In order to deal with this issue, an additional parameter may be introduced to the system such as ρ that represents the replication factor.

Replication factor parameter designates how much each portion of the execution history is replicated among replicas. When performing history saving decision, each replica checks the replication factor parameter as well to starts saving history. For instance, for the straightforward example above ρ is 1 since each portion of the history is saved by a single replica. If we designate ρ as 2 then each portion should be saved by two replicas.

[2]We use simple brackets to represent an ordered set.
[3]We assume replica id's start from 0 in the context of this paper

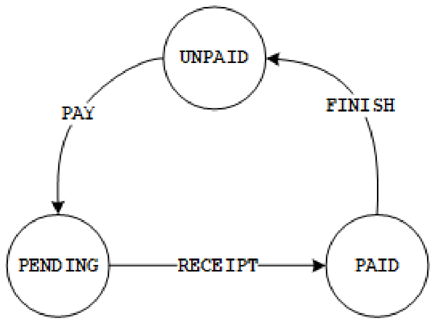

Fig. 4.1: Example book store state machine.

Parameter ρ can be adapted as in Eq. 3.4 simply by adding ρ number of additional condition where the right hand side of the equation is incremented once for each additional condition.

$$\bigwedge_{k=0}^{\rho} (r_{id} == ((e_i \;\texttt{div}\; \tau) \;\texttt{mod}\; |R|) + k) \tag{3.4}$$

For instance, for a 240 event execution, if the number of replicas is 6, τ is determined as 20 and ρ is determined as 2, replica with id 4 is going to save history for twenty events beginning from 80th, 100th, 200th and 220th events. Additionally, replica with id 5 is going to save history for twenty events beginning from 100th, 120th, and 220th events. Since we assume the execution history consists of 240 events, replica 5 is going to stop saving at the end of 240th event but for a case with longer execution histories it is going to continue saving for 20 events starting from 240th event as well. This approach is similar to mirroring and striping approach used in RAID 1+0 implementations [37].

A final remark might be to note that ρ parameter should not exceed the number of replicas, naturally. This parameter provides full history saving by all the replicas when it is set to the number of replicas and provide minimal level of reliability when it is set to 2. If ρ parameter is set to 1, distributed checkpointing will be useful only for the joining replicas to the replica cluster but it will be unreliable in case of a replica failure. It should also be noted that as ρ gets larger it will produce more overhead on each replica during checkpointing and recovery operations.

4. Experimental Environment.

4.1. Overall Architecture. We use a simple book store state machine, as shown on Fig. 4.1 to carry out experiments on distributed checkpointing approach. When a book is ready to be bought from customers, it starts with `UNPAID` state and waits the `PAID` event to be triggered. When the booking and payment operations are performed on the book, `PAID` event is triggered and state has been changed from `UNPAID` to `PENDING` 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, `RECEIPT` event is triggered and state is transited from `PENDING` to `PAID`. In our experiments we use an implementation of this state machine using Spring State Machines.

We set an experimental environment up using containers and a message queue as illustrated in Fig. 4.2. Each state machine is implemented using Spring State Machine framework inside containers running replicas of the book store state machine. Coordinator node is responsible from generating workload for state machine replicas and coordinating booting sequences of the state machine instances by communicating with replica agents through a simple message queue. We also use coordinators and agents to collect information about the run-time measures that we use to evaluate the performance of the experimented approaches. We have used this architecture in our local experiments as well as cloud experiments.

A typical execution of an experiments kicks off with booting all the replicated state machines inside current cluster. During their booting sequence, replicas prepares themselves to process events -initialize local and shared

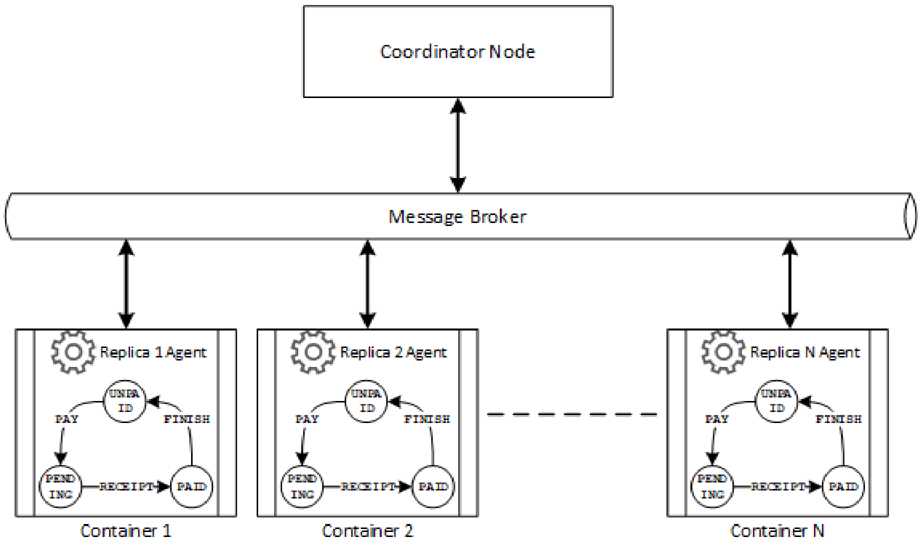

Fig. 4.2: Overall architecture.

variables and so on-, and then, begin listening to incoming events in order to perform transitions between state machine states. Controller node, creates necessary events and sends them to replicas via message broker. The message broker in our architecture is responsible from the following actions:

- Event communication between the coordinator node and replicas,
- Acknowledging controller node of replica life cycle,
- Communicating checkpointing information between replicas and the coordinator node,
- Measuring and reporting number of messages passed through during event processing.

Once the first event is send to state machines, it processes this event, performs necessary operations on its variables and finishes its execution in order to process a new event. After event processing finishes checkpointing operations are performed. During our experiments we use replicas to store checkpointing information as well as coordinator node whenever an external entity is necessary for the checkpointing approach. The details of the checkpointing approaches we have implement is explained in more detail in Sect. 4.2. During checkpointing, we store the context of the state machine which involves inputs and outputs of the current state. Inputs consist of incoming event, event timestamp, source state of the state machine while outputs consist of local and shared variables, destination state of the state machine. As a result of checkpointing process in each replica, whole execution history is recorded as a sequence of state machine contexts in replicas and/or coordinator node depending on the applied approach.

4.2. Implemented Approaches. For our experiments we implemented four different approaches to compare the performance of the distributed checkpointing approach. Initially we implemented centralised checkpointing approach where checkpointing information is stored only in the controller node. Afterwards we implemented conventional approach where each replica stores all the checkpointing information. Finally we implemented two variants of our approach: a striped DCfRSM where each replica stores a single portion of the execution history ($\rho = 1$) and a striped and mirrored DCfRSM where each replica stores two portions of the execution history ($\rho = 2$).

In case of centralised approach, none of RSMs store any of the checkpoints; instead all the checkpoint messages are stored by the controller node, ensuring all the events processed by all the replicas are persisted. To avoid the controller node to be a single point of failure, centralised node can also be replicated. We have left implementation and performance evaluation of such a scenario for a future study.

One of the checkpointing approaches that is used for benchmarking DCfRSM approach is conventional checkpointing. In this approach all the replicas perform checkpointing after each and every event processing, storing exactly the same checkpoints information. In case of any failure on any of the RSM instances, all the checkpoint information should be gathered and applied in order to join back to the RSM cluster. Likewise, a freshly booting replica should communicate with a/some running replica(s) to gather checkpoint information. Memory overhead of this approach is expected to be larger than other approaches, since checkpoint snapshots are redundantly stored by replicas.

As explained in Sect. 3, we implemented two variants of DCfRSM, with replication factor(ρ) set to one and two respectively, to reason about the amount of increase in the overhead as the replication factor parameter gets higher. As discussed earlier, employing a higher level of ρ provides more reliable checkpointing and a higher overhead to replicas.

We provide our implementations for the book store state machine[4] and controller node[5] openly hosted in a cloud repository service to make our experiments reproducible by the scientific community.

5. Experimental Evaluation. For the executions of tests, we use two different environments, a local cluster and a cloud based environment. All the experiments are conducted with 4, 6, 8 and 10 replicas in the experimental environment over 10 repetitions for each experiment by sending a total number of 3600 requests for the master state machine of the replicated cluster. We set the replica's history portion interval τ to 120 events being a common multiple for each different number of replicas used in the experiments and also being a divisor of total number of requests used in the experiments.

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. In order to measure memory consumption of each replica during state machine execution, an external library is used for counting number of checkpoint objects in memory. Java's instrumentation API[6] is used during state machine execution to measure and log memory usage whenever a checkpoint is about to be persisted. An overview of the application of our experiments can be summarised as follows:

- Initialise controller node and message broker,
- Initialise the necessary number of replica in the cluster,
- Trigger messages from controller node, wait for replicas to finish execution,
- Once all the events are processed, compare local and shared variables of all the replicas in order to ensure that replicas executed consistently,
- Boot a new replica in order to join the cluster, wait for the replica to gather checkpoint information from respective node/nodes

Once the new replica finishes 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 for the replicas. Then, averages and standard errors for repeated experiments are calculated.

5.1. Experiments on local cluster. As a local cluster we use computers with 2.60 GHz Intel i5 processors, 4 GB RAM and 100 GB SSDs running debian linux distributions. We begin presenting the experiments on our local cluster by examining memory consumption of each approach on average for each replica. Figure 5.1 presents and compares the memory consumption for approaches. As expected, conventional approach constantly consumes the highest amount of memory since all the replicas in the cluster keep the whole history all the time for this approach. Likewise, centralised approach constantly consumes the lowest amount of memory since replicas do not keep any checkpointing history for this approach. Distributed checkpointing approaches stand in the middle between conventional and centralised approaches and spend less memory as the number of replicas increase since the history will be divided among more number of replicas. Comparably, mirroring on top of striping increases the amount of memory consumption, as expected.

For restore duration in Fig. 5.2, the results in the local cluster are close and have high deviations. However,

[4]`https://github.com/celikelozdinc/DistributedStateMachine`
[5]`https://github.com/celikelozdinc/LoadBalancer`
[6]`https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html`

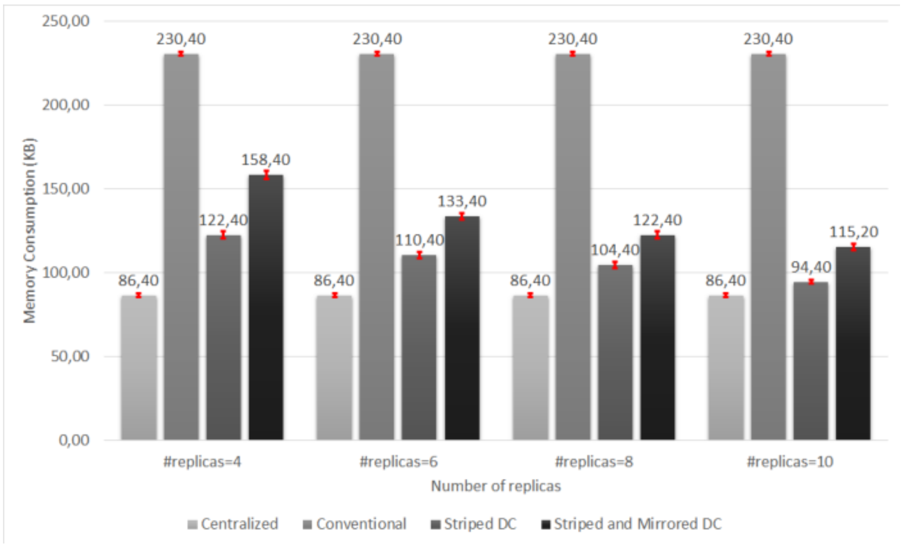

Fig. 5.1: Average memory consumption by replicas.

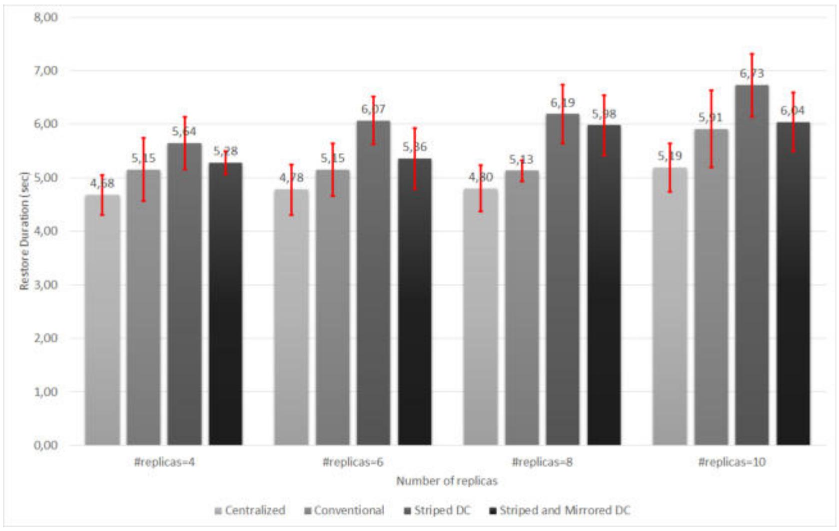

Fig. 5.2: Restore duration.

the average restore duration for centralised approach performed the best with respect to other approaches since the booting sequence requires less communication and only with coordinator node. Conventional approach came the second because obtaining history information from another replica requires and extra step of communication in our implementation compared to centralised approach: during booting sequence checkpoint information needs to be communicated from a running replica to coordinator and then from the coordinator to booting replica. This overhead may be avoided by enabling the booting replica to directly obtain checkpoint data from a running replica. Distributed checkpointing approaches perform worse because they need more communication

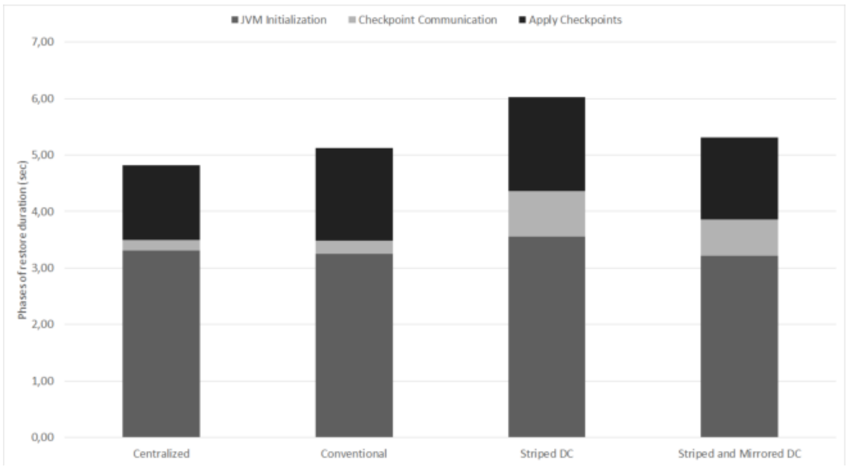

Fig. 5.3: Average time spend for each phases of restoring

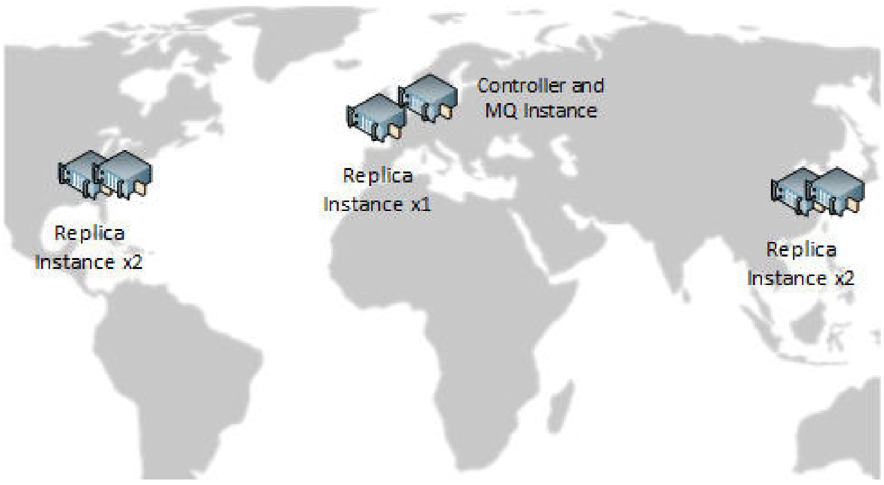

Fig. 5.4: AWS EC2 architecture for cloud experiments.

with other replicas more than the conventional and centralised approaches. An important remark might be the mirrored and striped approach beating the striped approach for restore duration. We believe, this is due to the more number of alternatives to obtain checkpoint data portions during restoring phase. Any slow responding, bottleneck, replica is eliminated due to the presence of alternatives to obtain the same data when striping and mirroring is applied.

Furthermore, we investigated the time spent during the restoring of a booting replica in Fig. 5.3. It can be seen that the difference between different approaches is greatly due to the checkpoint data communication phase.

5.2. Experiments on cloud environment. We also repeat our experiments on geographically distributed `t3.medium` instances running on a Amazon Web Services Elastic Compute Cloud (AWS-EC2). As per Fig. 5.4, 6 virtual machines from 3 different regions are used for executing experiments in cloud environment. While executing experiments, controller node and message broker service is isolated and positioned on a dif-

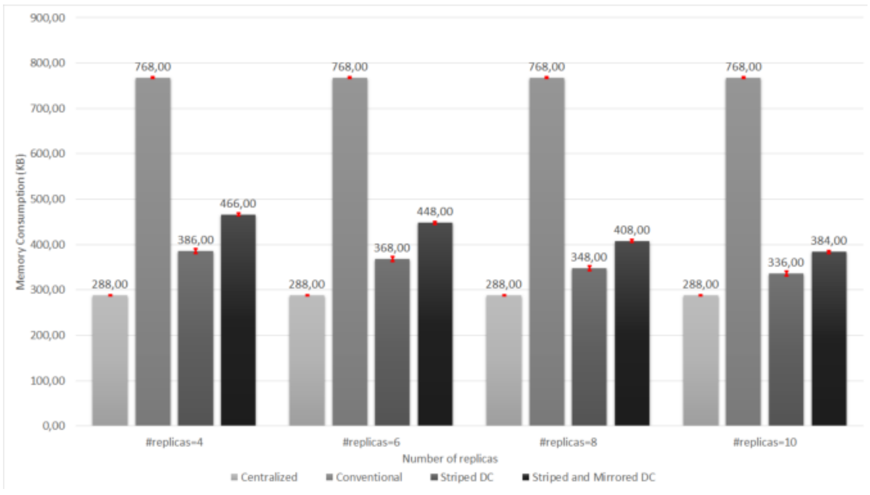

Fig. 5.5: Memory consumption from the experiments in cloud.

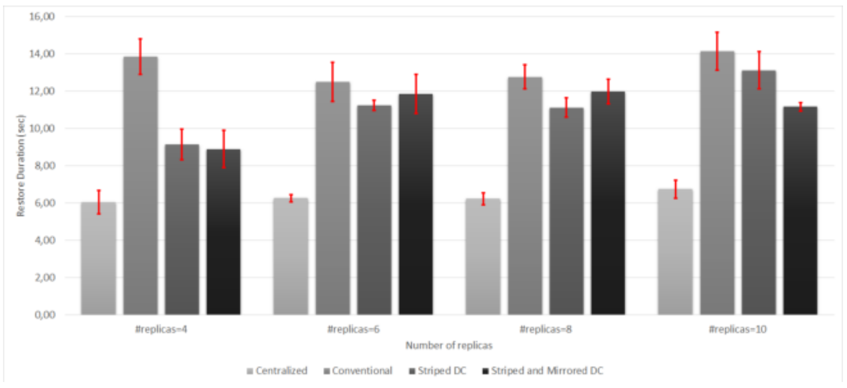

Fig. 5.6: Restore duration from the experiments in cloud.

ferent region which is totally apart from RSM instances. All the RSM instances distributed among 4 virtual machines and hence, spread to 2 regions. Freshly booting replicas are joint to cluster from a region apart from the regions of active replicas. By doing so, whenever a new replica joins the cluster, it is needed to gather checkpoint snapshots from different machines on geographically distributed regions.

Figure 5.5 shows the same advantage of DCfRSM approach in terms of memory consumption. Distributed approaches spend less memory since they divide the history data to multiple parts during their execution. Centralised approach is the best in this respect, naturally, since it doesn't require any replica to keep any checkpointing data.

As per Fig. 5.6, results from experiments in cloud environment shows some differences in terms of restore duration in cloud environments. Conventional approaches perform worse than the rest due to the fact that the booting replica is always in a geographically different region than the replica that provides checkpointing information. Likewise, for the centralised approach it is guaranteed that the booting replica and the coordinator are in the same region, which provides an advantage of communication latency during the booting time. In more realistic scenarios these measurements might change form case to case. A booting replica might not always find the checkpoint data in a close replica in terms of geographical location or network latency. However, our experiments provide the best and worst case scenarios under centralised and conventional approaches respectively to show the place of distributed checkpointing approaches with this respect. For distributed checkpointing, restore duration is always better than conventional approach even though the booting replica is in a different region than all the other replicas. This situation is due to the exploitation of alternatives instead of relying on a single replica. On the other hand, for cloud experiments, striped approach has performed slightly better than the striped and mirrored approach for small number of replicas but striped and mirrored approach performed much better as the number of replicas reached to 10. This situation shows that for increased network latency mirroring might lose its positive effect on restore duration for small number of replicas.

6. Conclusion and Future Work. In this paper, distributed checkpointing for the replicated state machines is examined and compared with conventional approaches. Especially in terms of full replication of checkpointing data and using a single node, distributed checkpointing approaches provide a mediation point to leverage between the amount of fault tolerance of the cluster versus restore duration of the replicas. Our experiments show that using distributed checkpointing provides a certain amount of memory consumption advantage and provides worse (as expected) but comparable restore duration. Main advantage of using a distributed checkpointing approach is to distribute the checkpointing information among replicas to provide an adjustable level of fault tolerance during replicated state machine execution.

Our studies can be extended to decrease recovery time overhead as much as possible in order to provide a better trade-off between distributed checkpointing and other approaches. Though many possible improvement opportunities exist in our implementations, allowing replicas to communicate each other via agents to eliminate the need to use a coordinator node might the most important one. Another possibility might be to better parallelise the recovery phase for distributed checkpointing since the approach benefits from using independent portions of the execution history. Moreover, various different values for parameters τ and ρ might be used to find optimal values for different scenarios in RSM checkpointing. Finally, providing the reliability of history portions by using parity information instead of mirroring might be another possible improvement to store even less checkpointing information in this context.

REFERENCES

[1] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, R. Rodrigues, Making geo-replicated systems fast as possible, consistent when necessary. *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 265-278, 2012.

[2] W.J. Bolosky, D. Bradshaw, R.B. Haagens, N.P. Kusters, P. Li, Paxos replicated state machines as the basis of a high-performance data store. *Proc. NSDI'11, USENIX Conference on Networked Systems Design and Implementation*, pages 141-154, 2011.

[3] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, F. Pedone, Clock-RSM: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 343-354, IEEE, 2014.

[4] N.Ö. Çelikel, T. Ovatman, A Distributed Checkpoint Mechanism for Replicated State Machines. *Proceedings of the 10th International Conference on Cloud Computing and Services Science, CLOSER 2020*, Prague, Czech Republic, May 7-9, 2020, pages 515-520, SCITEPRESS, 2020.

[5] Posner, J. System-Level vs. Application-Level Checkpointing. *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 404–405, 2020.

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

[7] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, A. Warfield, Remus: High availability via asynchronous virtual machine replication. *Proceedings of the 5th USENIX symposium on networked systems design and implementation*, pages 161-174, San Francisco, 2008.

[8] J. Heo, S. Yi, Y. Cho, J. Hong, S.Y. Shin, Space-efficient page-level incremental checkpointing. *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1558-1562, 2005.

[9] W. Zhao, Performance optimization for state machine replication based on application semantics: a review. *Journal of Systems and Software*, 112:96-109,2016.

[10] F.B. Schneider, Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*,

[11] Guerraoui, R., Knežević, N., Quéma, V., Vukolić, M., The next 700 BFT protocols. *ACM Transactions on Computer Systems*, Vol. 32, No. 4, pages 12:1–12:45, 2015.

[12] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E., Zyzzyva: speculative byzantine fault tolerance. *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages=45–58, 2007.

[13] O.M. Mendizabal, Fast recovery in parallel state machine replication. *Pontifícia Universidade Católica do Rio Grande do Sul*, 2016. 22(4):299-319, 1990.

[14] Cao, J., Simonin, M., Cooperman, G., Morin, C., Checkpointing as a service in heterogeneous cloud environments. *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 61–70, 2015.

[15] Zhou, A., Sun, Q., Li, J., Enhancing reliability via checkpointing in cloud computing systems. *China Communications, IEEE*, volume 14, number 7, pages 1–10, 2017.

[16] Mirhoseini, A., Rouhani, B. D., Songhori, E., Koushanfar, F. Chime: Checkpointing long computations on interm ittently energized iot devices. *IEEE Transactions on Multi-Scale Computing Systems*, volume 2, number 4, pages 277–290, 2016.

[17] Meroufel, B., Belalem, G., Optimization of checkpointing/recovery strategy in cloud computing with adaptive storage management. *Concurrency and Computation: Practice and Experience*, volume 30, number 24, pages e4906, Wiley Online Library, year=2018.

[18] Teylo, L., Brum, R. C., Arantes, L., Sens, P., Drummond, L. M. D. A., Developing Checkpointing and Recovery Procedures with the Storage Services of Amazon Web Services. *49th International Conference on Parallel Processing-ICPP: Workshops*, pages 1–8, 2020.

[19] Levitin, G., Xing, L., Luo, L., Joint optimal checkpointing and rejuvenation policy for real-time computing tasks. *Reliability Engineering & System Safety, Elsevier*, volume 182, pages 63–72, 2019.

[20] M. Yanhua, P.J. Flavio, M.Keith, Mencius: building efficient replicated state machines for WANs. *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, 2008.

[21] J. Konczak, N.F. de Sousa Santos, T. Zurkowski, P. Wojciechowski, A. Schiper, JPaxos: State machine replication based on the Paxos protocol. *EPFL- I&C - School of Computer and Communication Sciences, LSR - Distributed Systems Laboratory, Technical Report*, No. REP_WORK, 2011.

[22] B. Ghit, D. Epema, Better safe than sorry: Grappling with failures of in-memory data analytics frameworks. *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 105-116, 2017.

[23] N. Naksinehaboon, Y. Liu, C. Leangsuksun, R. Nassar, M. Paun, S.L. Scott, Reliability-aware approach: An incremental checkpoint/restart model in hpc environments. *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 783-788. IEEE, 2008.

[24] J.G. Slember, P. Narasimhan, Static Analysis Meets Distributed Fault-Tolerance: Enabling State-Machine Replication with Nondeterminism. *HotDep*, 2006.

[25] J.C. Sancho, F. Petrini, G. Johnson, E. Frachtenberg, On the feasibility of incremental checkpointing for scientific computing. *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings*, page 58, IEEE, 2004.

[26] R. Gioiosa, J.C. Sancho, S. Jiang, F. Petrini, 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*, pages 9-9, IEEE, 2005.

[27] Zhang, W., Shenker, S., Zhang, I., Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1029–1046, 2020.

[28] Ahmed, S., Bhatti, N. A., Alizai, M. H., Siddiqui, J. H., Mottola, L., Efficient intermittent computing with differential checkpointing. *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages=70–81, 2019.

[29] Jayasekara, S., Harwood, A., Karunasekera, S. Optimal Multi-Level Interval-based Checkpointing for Exascale Stream Processing Systems. *arXiv preprint arXiv:1912.07162*, 2019.

[30] Ren, K., Diamond, T., Abadi, D. J., Thomson, A., Low-overhead asynchronous checkpointing in main-memory database systems. *Proceedings of the 2016 International Conference on Management of Data* pages 1539–1551, 2016.

[31] Akturk, I., Karpuzcu, U. R. ACR: Amnesic Checkpointing and Recovery. *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 30–43, 2020.

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

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

[34] J. Rushby, Reconfiguration and transient recovery in state machine architectures. *Proceedings of Annual Symposium on Fault Tolerant Computing*, pages 6-15, IEEE, 1996.

[35] F.B. Schneider, L. Zhou, Implementing trustworthy services using replicated state machines. *IEEE Security & Privacy*, 3(5):34-43, 2005.

[36] B. Wester, J.A. Cowling, E.B. Nightingale, P.M. Chen, J. Flinn, B. Liskov, Tolerating Latency in Replicated State Machines Through Client Speculation. *NSDI*, pages 245-260, 2009.

[37] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, D.A. Patterson, RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145-185, 1994.

Edited by: Dana Petcu
Received: Dec 3, 2020
Accepted: Jan 21, 2021