A Distributed Checkpoint Mechanism for Replicated State Machines

Niyazi Özdinç Çelikel[1] and Tolga Ovatman[2][a]
[1] *Siemens R&D Center, Istanbul, Turkey*
[2] *Department of Computer Engineering, Istanbul Technical University, 34469 İstanbul, Turkey*

Keywords: Replicated State Machines, Distributed Checkpoints, Fault Tolerance.

Abstract: This study presents preliminary results from a distributed checkpointing approach developed to be used with replicated state machines. Our approach takes advantage from splitting and storing partial execution history of the master state machine in a distributed way. Our initial results show that using such an approach provides less memory consumption for both the running replicas and the restoring replica in case of a failure. On the other hand for larger histories and larger number of replicas it also increases the restore duration as a major drawback.

1 INTRODUCTION

Increasing popularity of serverless software hosting services offered by major cloud vendors boosted the usage of state machine models in various areas of software development for cloud. Serverless computing offers software developers to eliminate the necessity to take care of many layers/aspects of software development stack and focus on developing and expressing tasks to be executed by cloud system. It is possible to use different expression models; however, using state machine models to express the functionality is one of the frequently preferred approaches for serverless computing.

As well as the independence form software stack, serverless computing also offers non functional requirements such as automatic scaling, fault tolerance, etc.. Replicating a master state machine that is going to work in a distributed way to handle and synchronously process requests is a widely used model to provide fault tolerance. Naturally, this approach has its own challenges and limitations. A major approach that can be used in fault tolerance is checkpointing. Checkpointing involves saving the system state in periodic intervals to boot the system back in time of a failure. Application of checkpointing in replicated state machines and increasing the effectiveness of the process is an active research area in today's cloud computing systems.

In this study we investigate the idea of distributing the state machine snapshot images among the state machine replicas to decrease the amount of memory overhead used by each replica when saving the system state. Our approach involves striping the request history into many pieces during the checkpointing process so that each replica is going to store a certain piece of the history while the whole history can be obtained from the overall system.

We have implemented our approach using Spring state machine framework and compared our results with the default mechanism which Spring SM uses to provide fault tolerance for the state machine ensembles. By default Spring SM stores the whole history in the ensemble communication medium among the state machines, which is used as an Apache Zookeeper instance in our case. This makes the Zookeeper system a single point of failure, whenever a failure occurs in Zookeeper system loses the checkpoint snapshots that has been taken so far.

Our preliminary results show that our approach eliminates this single point of failure by distributing the checkpoint snapshots among the replicas. By controlling the amount of redundancy, distributed snapshots can be gathered even if more than one replica fails at the same time, during executions. Moreover we also decrease the amount of memory overhead needed for the checkpoint mechanism by splitting the snapshot information among the replicas. Currently, our approach, in its preliminary state produces a heavy recovery-time overhead since the failed replica needs to collect pieces from different replicas in order to obtain a full history of execution. Such a trade-off is unavoidable but our future study is aimed

[a] https://orcid.org/0000-0001-5918-3145

Çelikel, N. and Ovatman, T.
A Distributed Checkpoint Mechanism for Replicated State Machines.
DOI: 10.5220/0009797405150520
In *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, pages 515-520
ISBN: 978-989-758-424-4

towards minimizing this overhead.

In the rest of the paper, we begin by summarizing the related work. Afterwards we explain our distributed checkpointing mechanism and provide an exemplary architecture and experimental environment to evaluate out approach in Section. We finalize with conclusions and planned future work.

2 RELATED WORK

Performance of the checkpointing has always been a widely investigated topic in the distributed computing domain. There are numerous techniques to be used as a solution in order to minimize checkpoint and recovery costs. (Heo, Junyoung & Yi, Sangho & Cho, Yookun & Hong,Jiman & Shin,Sung, 2006) proposes a solution for minimizing checkpointing costs in terms of storage aspects while (Mao, Yanhua and Junqueira, Flavio and Marzullo, K., 2008) focuses on high network performance, in terms of throughput. (Mao, Yanhua and Junqueira, Flavio and Marzullo, K., 2008) propose a high-performance replicated state machine check-pointing and recovering approach derived from Paxos consensus protocol, which is out of scope for this research. (B. Ghit and D. H. J. Epema, 2017) propose to checkpoint only straggling tasks in order to minimize the number of checkpoints and hence, overall checkpointing overhead. (Naksinehaboon, Nichamon and Liu, Yudan and Leangsuksun, C. and Nassar, Ruba and Paun, Mihaela and Scott, S., 2008) propose a novel checkpointing mechanism in order to reduce the checkpoint data size by checkpointing only dirty pages that are modified since last checkpoint time. This novel approach named as incremental check-point model which involves a decision mechanism in order to persist the minimal necessary data to be check-pointed since the last checkpoint time in the execution history.

There are many other efforts which is targeted for finding a way to efficiently implement the checkpointing mechanism in system level (Gioiosa, R. and Sancho, J.C. and Jiang, S. and Petrini, Fabrizio, 2005) or user level (Sancho, J.C. and Petrini, Fabrizio and Johnson, G. and Frachtenberg, Eitan, 2004). (Sancho, J.C. and Petrini, Fabrizio and Johnson, G. and Frachtenberg, Eitan, 2004) states the user level approach as checkpointing is performed explicitly by external applications and propose the approach for determining optimal checkpoint frequency as a matter-of-fact. (Gioiosa, R. and Sancho, J.C. and Jiang, S. and Petrini, Fabrizio, 2005) defines the system-level approach as generally-applicable approach, which can be defined as an application is unaware whether it is checkpointed or not. Gioiose et al. also proposes an innovative methodology called buffered co-scheduling which is implemented at kernel level, hence has unrestricted access to processor registers, file descriptors, and states several check-pointing formulations to be used, such as internal check-pointing in which uses UNIX/LINUX signal mechanism.

The idea behind using replicated state machines in order to model distributed check-pointing approach is already stated by (Bolosky, William and Bradshaw, Dexter and Haagens, Randolph and Kusters, Norbert and Microsoft, Peng, 2011) and (Fred B. Schneider, 1990), replicated state machines can be made fault-tolerant by running on multiple computers with feeding the same inputs.

3 DISTRIBUTED CHECKPOINTING

Distributed checkpointing approach for replicated state machines utilizes the idea of each replica saving the state of execution history for a predesignated period of time. This way each replica stores one or more portions of the execution history locally, later to be retrieved by a freshly booting replica.

Let's assume that each state machine consists of some states denoted as s_i and some actions that trigger transition between states such as $s_i \xrightarrow{a_k} s_j$ where transition from state i to state j is triggered by action k. These definitions result in a basic state machine model in 1 that is going to be used in this paper's context. In this definition δ defines the transitions as a function from state-action pairs to states.

$$\begin{aligned} M &= \{S, A\} \\ S &= \{s_0, s_1, \ldots\} \\ A &= \{a_0, a_1, \ldots\} \\ \delta &= S \times A \to A \end{aligned} \tag{1}$$

Regarding the definition in equation 1 we can exemplify the execution history for a state machine as in definition in equation 2 where the history begins with a state and continues by action-state pairs where each state is navigable by the related action in the state machine definition. This example can be used to represent portions of history where the history may contain only a portion of the full execution of the state machine. However if the history begins by the initial state (e.g. s_0) than the history represents the full execution history until the final state of the sequence in the history.

$$H = s_i, a_j, s_k, a_p, s_r, \ldots \tag{2}$$

The definition in equation 2 does not include any temporal information about the execution history other than the sequence of state transitions. To define the time interval that the history belongs to we are going to use a superscript to indicate a discrete clock tick interval that begins from 0 tick as the full execution's beginning time. In addition, we are going to use a subscript to associate a history with the id of a replica which currently stores the given partial history.

$$\begin{aligned} H^{0-39} &= s_0, a_0, s_1, a_1, s_2, a_2, \ldots, s_{19}, a_{19}, s_{20} \\ H_0^{0-9} &= s_0, a_0, s_1, a_1, \ldots, a_4, s_5 \\ H_1^{10-19} &= s_5, a_5, s_6, a_6, \ldots, a_9, s_{10} \\ H_2^{20-29} &= s_{10}, a_{10}, s_{11}, a_{11}, \ldots, a_{14}, s_{15} \\ H_3^{30-39} &= s_{15}, a_{15}, s_{16}, a_{16}, \ldots, a_{19}, s_{20} \end{aligned} \tag{3}$$

For instance, the full history indicated as H^{0-39} in equation 3 describes an execution history collected between time ticks 0 and 39, where 21 different states has been travelled, triggered by 20 different actions. In order to be more illustrative all the actions and states are chosen as distinct and numbered in sequence for the sake of this example. In a realistic scenario there would have been arbitrary transitions among the states of the state machine with arbitrary incoming actions. In equation 3 the full history is kept by four different replicas, each indicated with H_i^{s-e} where i represents replica id, s represents beginning time tick and e represents end time tick. The history is divided into four different partial histories in this example where each part is kept in a different replica.

In a replicated state machine, logically all the replicas act as a logical master state machine where any request, triggering an action, received by a replica is synchronously communicated and reflected to other replicas, causing the other replicas to execute the same action in their local machines. This situation results is a logical master history, which corresponds to H^{0-39} in our example, be shared between all the replica state machines.

In our distributed checkpointing, approach we have implemented the underlying mechanism for the replicated machines to synchronously store, collect and merge this logical history by the currently running replicas. Since there is a single logical history for all the replicas, it is not required to store a real time clock to perform fair sharing of the execution histories; instead it is sufficient to count the number of actions in current partial history to decide when to start/stop storing a partial history by each replica.

Briefly, our distributed checkpointing approach utilizes the discrete time depending on executing actions in replicated state machines to distribute the checkpoint storing responsibility between the current members of an ensemble (replica group). In this paper we investigate the advantages and overhead of performing such a distribution in a replica group. We can perform many additional tasks such as re-distributing histories to load balance for replicas that has just joined an ensemble or storing redundant partial histories to eliminate single point of failure in a replica group.

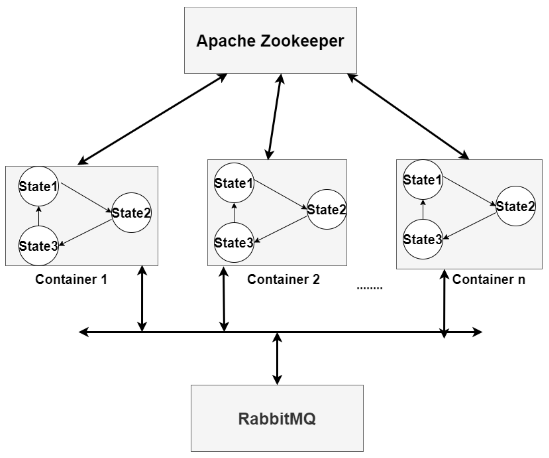


Figure 1: Overall Architecture.

4 SYSTEM ARCHITECTURE AND EXPERIMENTAL ENVIRONMENT

In our system architecture illustrated in Figure 1, replicated state machines perform simplified operations on its own local variables, such as incrementing and decrementing, and also updates shared variables while passing from one state to another. Once all the replicated state machines are started to run, they waits events to be triggered in order to perform transitions between states. When 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 the incoming event. Once the processing of the event is finished, all the state machines will be in the same state and then will wait next event to process. The synchronization between states is established by master machine.

In our experimentation environment base functionalities used for executing distributed checkpointing mechanism are built. There are three clients which executes replicated state machines in Figure 2

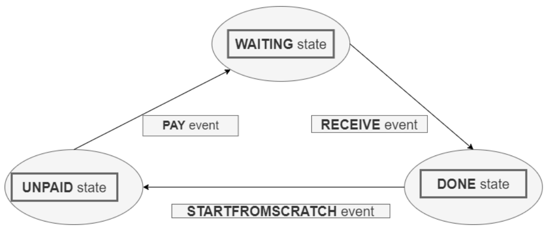


Figure 2: States and transitions of the replicated state machines.

and one master machine which hosts the Apache Zookeeper binaries and provides coordination mechanism for replicated state machines.

Each of these clients and master machine runs as containers in Docker Engine. The aim of using docker containers is simplifying the connections between the clients and master in an isolated network structure. Our proposed mechanism do not need to be run inside containers; it can be applied to any replicated state machine implementation. In our experimental implementation we have chosen to use containers to host the replicas. Moreover, by using containerization phenomenon, clients and master can be started up in repetition by using minimal resources of the machine which hosts docker daemon. All the clients and master machine runs as lightweight linux containers. In addition to that, with the aim of starting and stopping containers, starting and stopping replicated state machines inside containers, docker-compose tool is used for orchestration purposes.

An example docker-compose file is provided in order to start replicated state machines inside containers:

```
version: "3"
services:
  smoc1:
    build:
      context: .
      dockerfile: smoc/Dockerfile
    networks:
      - distributedWan
    hostname: ${HOSTNAME_SMOC1}
    environment:
      - EXCHANGE=${IPC_EXCHANGE_FOR_SMOC1}
      - QUEUE=${IPC_QUEUE_FOR_SMOC1}
  smoc2:
    build:
      context: .
      dockerfile: smoc/Dockerfile
    networks:
      - distributedWan
    hostname: ${HOSTNAME_SMOC2}
    environment:
      - EXCHANGE=${IPC_EXCHANGE_FOR_SMOC2}
      - QUEUE=${IPC_QUEUE_FOR_SMOC2}
networks:
  distributedWan:
    external:
      name: loadbalancer_isolatedNetwork
```

Distributed checkpointing is continuously being performed during the life-cycle of the replicated state machines. Each of the state machines being executed in each clients, does not store whole execution history -which includes state transitions, states, local and shared variables-, they only store minimal portion of the execution history. Once an event is forwarded to any of replicated state machines inside the ensemble, it is stored locally by the state machine which processed this event on its local context. Context for the replicated state machines can be modelled as *input* -incoming event, event timestamp, source event- and as *output* -local and shared variables, destination state- and be stored as a whole. Hence, the replica which processes the event is responsible for storing related artifacts regarding this event. As a result of checkpointing process in each client, whole execution history is persisted in distributed manner in different containers.

In case of any failure in any of replicated state machines, execution history can be gathered from non-faulty clients via a broadcast message.Communications between client machines which hosts replicated state machines are established via *remote procedure calls* using the *RabbitMQ* message broker tool. Once the faulty client processed the incoming messages, sorted and applied in the executed order on its own state machine, it is ready to re-join the cluster again.

From that point on, experimental setup is presented. A 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. The communication medium among containers are established via network features of the docker-compose tool. The master node hosts the Zookeeper binaries whereas client nodes -which are based on *alpine* base image- serve as hosts for the replicated state machines.

5 EVALUATION

We have conducted experiments with 4, 8, 12 and 16 replicas in an experiment environment described in the previous section and measured the amount of difference between memory consumption and restore duration.

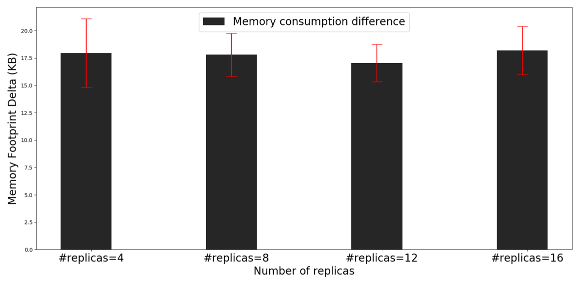
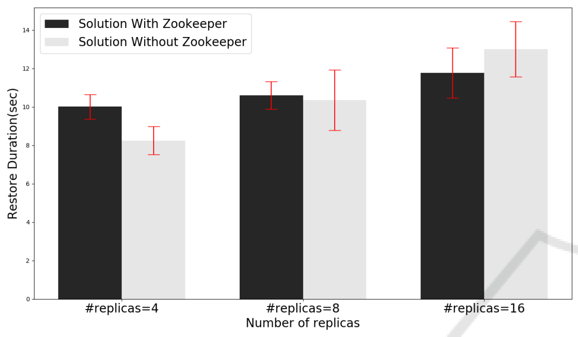
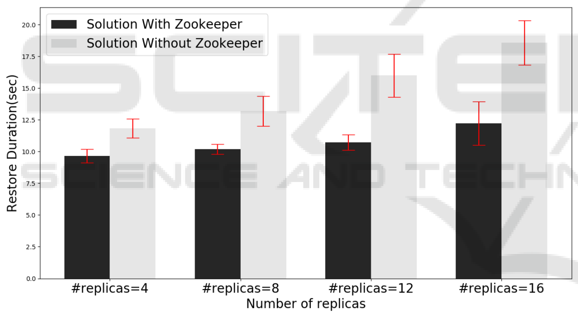


Figure 3: Memory consumption difference of the replica being restored.



(a) 96 requests



(b) 960 requests

Figure 4: Replica restore durations.

In Figure 3, we present the difference in memory consumption of the replica that is being restored with and without using distributed checkpoints. Each bar in the figure represents the difference of memory consumption between conventional checkpointing and distributed checkpointing. Distributed checkpointing mechanism provided around 17 kilobytes of advantage compared to the conventional checkpoint mechanism present in Spring state machine framework.

In Figure 4, we present the restore duration for a history with 96 different requests and 960 requests respectively. We have used a ten-fold input in terms of number of requests to observe the effect of large number of requests on restore duration during our comparison. It can be seen that even though our approach provides comparable results for small histories and small number of replicas as the amount of communication need increases our approach starts to perform poorly.

We would also like to note that using distributed checkpoints also decreased the memory consumption from 2 up to 7 kilobytes in each replica for the experiments above. Even though there exists an advantage in using distributed checkpoints in terms of memory consumption, still the trade-off for restore duration seems to be high for larger systems.

6 CONCLUSION AND FUTURE WORK

In this preliminary work we have proposed a distributed checkpoint mechanism that can be used in replicated state machines. Our work relies on splitting execution histories into multiple parts during the execution so that each part can be stored by another replica. This approach also allows redundant storage of partial histories to reduce the number of critical replicas that may render the system useless in case of failure. Besides the mentioned advantage, our approach provided less memory usage in present replicas and in the replica which is being recovered. In its current form, recovery time is increased because of the extra communication overhead needed for collecting partial histories from replicas. We plan to work on decreasing this overhead as much as possible to provide a better trade-off for distributed checkpointing. Another possible area of extension is balancing the amount of checkpoint data stored in each replica with the addition of new replicas to an existing ensemble.

ACKNOWLEDGEMENTS

This study is supported by the scientific and technological research council of Turkey (TUBITAK), within the project numbered 118E887

REFERENCES

B. Ghit and D. H. J. Epema (2017). Better safe than sorry: Grappling with failues of in-memory data analytics frameworks. *HPDC*, pages 105–116.

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*, pages 141–154.

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

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*, pages 9–9.

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

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*, pages 369–384.

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)*, pages 783–788.

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–.