Checkpointing Techniques in Distributed Systems: A Synopsis of Diverse Strategies Over the Last Decades

Henrique Goulart, Álvaro Franco, Odorico Mendizabal

[1] Programa de Pós-Graduação em Ciência da Computação (PPGCC)
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brasil

sgoulart.henrique@gmail.com,

alvaro.junio@ufsc.br, odorico.mendizabal@ufsc.br

Abstract. *This paper concisely reviews checkpointing techniques in distributed systems, focusing on various aspects such as coordinated and uncoordinated checkpointing, incremental checkpoints, fuzzy checkpoints, adaptive checkpoint intervals, and kernel-based and user-space checkpoints. The review highlights interesting points, outlines how each checkpoint approach works, and discusses their advantages and drawbacks. It also provides a brief overview of the adoption of checkpoints in different contexts in distributed computing, including Database Management Systems (DBMS), State Machine Replication (SMR), and High-Performance Computing (HPC) environments. Additionally, the paper briefly explores the application of checkpointing strategies in modern cloud and container environments, discussing their role in live migration and application state management. The review offers valuable insights into their adoption and application across various distributed computing contexts by summarizing the historical development, advances, and challenges in checkpointing techniques.*

1. Introduction

The development of distributed systems techniques has significantly influenced modern software. These techniques employ replication strategies, which ensure resilience and scalability by executing replicated services in parallel [Lamport 2019]. This approach enables replicated machines to handle multiple requests concurrently and addresses failures by maintaining multiple instances of the same services [Schneider 1990, Kotla and Dahlin 2004]. However, recovering machines and restoring their state after a failure is crucial; otherwise, all machines may eventually fail. To overcome this challenge, checkpointing techniques are employed, allowing the system to restore a machine from a more recent saved state instead of its initial state.

Checkpointing strategies are responsible for saving service snapshots, while recovery protocols restore a consistent state and restart processes or system components from these snapshots. Research and development of log and checkpoint-based recovery protocols are not new, reports from the 1970s already discussed the practical implementation of checkpoint/restart procedures in the third generation of computers [Oppenheimer and Clancy 1968, Chandy and Ramamoorthy 1972]. Even the IMS/360, recognized as one of history's pioneers and influential DBMS (Database Management Systems), implemented a checkpoint-based recovery functionality [McGee 1977]. Despite minor changes in the design of checkpoint and recovery protocols, all those precur-

sor approaches point out a substantial performance reduction and the high costs with the addition of storage capacity for saving checkpoints.

With technological advances in the following decades, computing platforms enabled the increase in processing and storage capacity and reduced latency in communication in computer networks. Powerful infrastructures and monetary reduction on systems development stimulated a broader adoption of checkpoint-based approaches to provide recovery and even other uses in distributed computing. For instance, checkpoint/recovery solutions support the development of Byzantine-fault tolerant services in the context of replication [Castro and Liskov 1999] and intrusion tolerance [Sousa et al. 2009]; they are at the core of migration strategies for virtualized platforms [Elmore et al. 2011]; are present in HPC as they provide fast resuming of long-running tasks in shared infrastructures with a limited period of use [Egwutuoha et al. 2013].

Despite a broad body of work regarding checkpoint-based techniques, there are just a few comprehensive reports about the advances in the field. In particular, the surveys focus on specific areas in the distributed systems, such as global checkpoints in message-passing systems [Elnozahy et al. 2002], or checkpoints in HPC [Egwutuoha et al. 2013]. This paper sheds some light on advances in checkpoint-based research over the last 50 years. We summarize the main advances in algorithms and design of checkpoint/recovery protocols and present a fresh view of the adoption of checkpoints in different contexts in distributed computing.

2. The Evolution of Checkpoint-Based Techniques

Checkpoint-based recovery augments system processes with the ability to persist executed operations, enabling processes to replay their execution after a failure beyond the most recent checkpoint. However, these techniques negatively impact the performance of the target applications [Bessani et al. 2013, Zheng et al. 2014, Mendizabal et al. 2016, Mendizabal et al. 2017]. For instance, synchronous writes for taking checkpoints can drastically reduce processes' throughput during regular operations. Although Solid State Devices (SSDs) can alleviate the problem, they can only partially solve it. In addition, checkpointing must interrupt the service during this operation, except when optimizations such as copy-on-write or fuzzy checkpoints are implemented. Next, we present some optimizations we found in the literature concerning reducing checkpoint overheads.

2.1. Coordinated and Uncoordinated Checkpoints

Coordinated checkpointing involves synchronizing all processes and taking a checkpoint of the entire system state to enable recovery in case of failures [Janakiraman and Tamir 1994, Chandy and Lamport 1985, Tamir and Sequin 1984]. In coordinated checkpointing, all processes must agree on a safe execution point in order to establish consistent checkpointing for all participants. In replicated systems, for example, synchronization can be guaranteed by consensus or total order broadcast protocols, which deliver the messages to all replicas in the same order. In a straightforward coordinated checkpointing approach, a designated process (the initiator) initiates the checkpoint by notifying all other processes of its intention to take a checkpoint. The two-phase checkpoint protocol is then used to ensure the consistency of the checkpoint. In the first phase of the protocol, each process enqueues the notification about the checkpoint and executes

all its previously enqueued operations. It then blocks itself and notifies the initiator that it is ready to take the checkpoint. The initiator waits until it receives notification from a quorum of processes that they are prepared to take the checkpoint. In the second phase of the protocol, the initiator sends another notification to each process to make the checkpoint of the entire system. Each process then takes a checkpoint of its local state. When processes finish taking their checkpoints, the initiator is notified and signals the processes to release their blocked state and return to regular operation. Coordinated checkpointing ensures that all processes take a consistent snapshot of the system state. On the other hand, it may cause a spike in latency time due to the time required to synchronize all processes and take the checkpoint globally.

In contrast to coordinated checkpointing, uncoordinated checkpointing allows processes to take checkpoints independently [Mostefaoui and Raynal 1996, Elnozahy et al. 2002, Mendizabal et al. 2014]. Each process captures its state without the knowledge of others, enabling them to take the checkpoint when it is more convenient and resume normal operation immediately after checkpoint completion. This method eliminates the need for process synchronization, allowing faster processes to proceed as soon as they complete the checkpoint. This approach reduces the chances of momentary unavailability or latency increases as it results in checkpoints taken at different times. However, uncoordinated checkpointing produces checkpoints with different states saved by the processes. Consequently, the recovery process becomes more complex, requiring awareness of these differences and the ability to manage them effectively.

An uncoordinated but deterministic checkpointing approach appears in [Bessani et al. 2013]. The authors present a sequential checkpointing strategy tailored for State Machine Replication (SMR). In this approach, replicas do not checkpoint their states simultaneously to avoid hiccups during normal execution. While one replica takes a checkpoint, other replicas process requests. Since SMR replicas make progress as long as a quorum of $n - 1$ replicas is available, there are f spare replicas in fault-free executions. The intuition is to make each replica store its state at different times to ensure that $n - f$ replicas can continue processing client requests.

2.2. Incremental Checkpoints

Incremental checkpoints provide a way to save only a portion of the system state instead of capturing the entire state for creating a checkpoint. This technique explicitly targets the system state components that have changed since the last checkpoint, reducing the checkpointing overhead [Elnozahy et al. 1992] and time needed to take the checkpoint. Despite the advantages of incremental checkpoints, they require the storage of multiple checkpoints, which could potentially increase storage usage. A coalescing or merging mechanism consolidates previous checkpoints and prevents indefinite growth. Additionally, it is important to note that incremental checkpoints are not suitable for every type of application, an application that frequently accesses and modify many different parts of the system will make the checkpointing process as costly as traditional checkpointing, capturing the entire system state, with the additional incremental checkpoint complexity. A prominent example of incremental checkpointing is Libckp [Plank et al. 1995], which efficiently saves only the modified pages of the Linux system since the last checkpoint by leveraging hardware capabilities to identify and isolate unchanged portions of the system state accurately.

2.3. Partitioned and Parallel Checkpoints

This approach aims to benefit from the parallelism in modern hardware, both in processing and I/O. The goal is to parallelize the checkpointing operation, potentially speeding up saving and restoring the checkpoint state. Parallel threads or tasks can manage separate state partitions of the target application. However, this approach introduces extra complexity to maintaining consistency, as the checkpoint states are divided into smaller partitions and do not represent the complete system state. Typically the recovery process becomes more complicated when dealing with partitioned checkpoints.

In [Junior 2020], authors proposed a checkpointing technique that allows the system to continue its regular execution in partitions not involved in checkpointing while a specific partition is undergoing the checkpointing process. A consequence is that a checkpoint does not represent a complete picture of the system state at a given time. These checkpoints with incomplete information are known in the literature as fuzzy checkpoints, and they impose new challenges on the recovery procedure.

2.4. Fuzzy Checkpoints

A noticeable side-effect of checkpoint execution is the system hiccups, demonstrating periods of unavailability or at least low throughput and latency increases. That happens because checkpoints typically must save a complete and consistent snapshot of the service state at a given point. One approach to reduce this overhead during normal operation is to allow the execution of the checkpointing procedure while regular operations are updating part of the service state. This approach benefits from parallelism but produces *fuzzy checkpoints*. This means that the resulting checkpoint image is not necessarily a consistent snapshot of the service state as of a particular point in the serial order of execution. Therefore, to recover a consistent snapshot, it is always necessary both to restore a checkpoint and to replay at least a portion of the log to perform the update of information that was not registered by the checkpoint.

In [Zheng et al. 2014], authors proposed a fast durability and recovery procedure for in-memory databases based on *fuzzy checkpoints*. Checkpoints may run in parallel with regular execution. Parts of the checkpoint are stored in multiple files, which simplifies the process of log truncation and take advantage of parallel I/O over multiple storage devices. A similar approach appears in [Junior 2020], where service replicas follow an active replication scheme and, at deterministic intervals, every replica saves the state of only a subset of the state partitions. When the workload is favorable, replicas can save distinct partitions, enabling checkpoint parallelism inter-replicas.

2.5. Dynamic and Adaptive Checkpoint Intervals

Traditional checkpointing methods typically employ fixed intervals to determine when a checkpoint should be taken. However, some studies have demonstrated that varying the interval periods can reduce the overall application computation time without affecting reliability. Considering probabilistic models and the specific knowledge about applications when deciding the instants to trigger a checkpoint procedure can further optimize the checkpointing process and improve system efficiency.

For example, one observation is that a failure is more likely to occur shortly after another failure [Tiwari et al. 2014]. Similarly, it is possible to consider the probability

of failure in certain contexts, as a process may only have a moderate likelihood of failure [Frank et al. 2021]. By adjusting checkpoint intervals based on the probability of failure, it is possible to reduce the overall computation time by avoiding unnecessary checkpointing and minimizing wasted computation, which refers to the computation lost between the last checkpoint and the moment of failure.

2.6. Process and System Level Checkpoints

Checkpoints can be implemented at the operating system kernel level or user space. Kernel-level checkpoints, like Linux Checkpoint/Restart as A Kernel module (CRAK) [Zhong and Nieh 2001] and the Berkeley Lab's Linux Checkpoint/Restart (BLCR) [Duell 2005], provide low-level access, enabling direct interaction with a process's data without difficulties. This approach allows for easier tracking of process states and the creation of consistent checkpoints. Furthermore, kernel-level checkpoints offer higher performance due to their direct operation within the operating system. However, since they are integrated into the kernel, they may require maintenance as the kernel evolves, potentially limiting code portability.

In user-space checkpointing [Duell 2005, Plank et al. 1995], the checkpointing process must monitor operating system signals to identify and track changes in memory regions, ensuring correct handling to achieve consistent checkpoints. Unlike kernel-level checkpointing, user-space checkpointing relies on listening to kernel interfaces and *system calls*, which may make it more susceptible to bugs and errors and potentially slower than the kernel-level approach. However, user-space checkpointing does not require kernel modifications, which enhances its portability across different systems.

3. Checkpoint-Recovery in Action

In the last decades, we could observe different uses for checkpoint/recovery strategies. Whether due to technological advances, which have increased storage and processing capacities and reduced network latency or due to the diversity of applications, which now demand reliable and highly available systems capable of recovering quickly. This section presents some relevant contexts where checkpoint-based recovery protocols are employed, although this is not an exhaustive exposition.

3.1. Rollback Recovery in Message-passing Systems

Many research works have investigated the side effects and challenges of developing consistent checkpoint-based recovery approaches for general-purpose distributed systems, including the comprehensive survey of checkpoint/recovery protocols in message-passing systems presented by [Elnozahy et al. 2002]. The paper defines the system model of message passing, in which different processes exchange messages with each other and the outside world (defined as something not controlled by the current system). Fault tolerance is achieved by storing the system's process state (checkpoints) in stable storage during regular application operation, allowing recovery from the saved state in case of failure.

In a system that interacts with the outside world and uses the checkpoint-based recovery-rollback protocol, a fault would require the system to inform the outside world to send all operations again since the last checkpoint was saved, allowing the system to re-execute those operations. This interaction with the outside world can be avoided using the

logging-based recovery-rollback protocol, which enables the system to replay the logged operations instead of requesting a resend from the outside.

Although the record of local logs stored by individual processes can turn interactions with the outside world aside, they raise some complexity and potential side effects when recovering from failures. As in *uncoordinated checkpointing*, processes make checkpoints independently without informing other processes. Each process may have multiple checkpoint files, one for each point in time. This approach can lead to issues like the *domino effect* caused by rollback propagation. When a process fails, and recovery is initiated, dependencies between processes may force non-failed processes to roll back to earlier states to maintain consistency. The problem is that rolling back one process may cause other processes to roll back, triggering a domino effect that can roll back the entire system to the initial state.

The issues caused by the domino effect can be avoided when checkpointing/recovery protocols track process dependencies to force the existence of *global states* throughout process execution. Such global states ensure consistency and are safe for recovery [Leu and Bhargava 1988]. Since *coordinated checkpointing* [Janakiraman and Tamir 1994] involves synchronizing all processes for taking a checkpoint, such strategies ensure that all processes take a consistent snapshot of the system state, which can be used for recovery in case of failures and avoids the domino effect.

Other researches also address scenarios where interacting processes in a distributed environment must reach a stable and consistent state. For instance, in [Chandy and Lamport 1985], authors propose a protocol where processes can determine a global state in the system during a distributed computation. Important problems can be cast in terms of the problem of detecting global states, for example, *computation termination*, *deadlock detection*, and, of special interest for this paper, *definition of a global and consistent state* among processes. As authors discussed, the famous Chandy-Lamport distributed snapshot algorithm is useful for implementing checkpoints suitable for rollback recovery in message-passing systems.

3.2. Database Management Systems

Database Management Systems (DBMS) commonly employ checkpointing strategies to ensure data consistency and fault tolerance. For example, the PostgreSQL[1,2] 15 database uses Write-Ahead Logging (WAL) to ensure that commands are first logged before being written to data files. This approach avoids the need to flush every state change operation to disk, allowing the system to operate with dirty pages while still maintaining fault tolerance. Since the log is written sequentially, it avoids random disk seeks, thereby improving performance. At checkpoint times, triggered either by a timeout interval or by the maximum allowed WAL size, all dirty pages must be flushed to the corresponding data files, and the checkpoint process is started. The I/O increase caused by this operation may degrade the system's overall performance. To mitigate this issue, PostgreSQL sets a target completion time for the checkpoint operation, spreading the I/O operations over time to avoid flooding the I/O system. After the checkpointing, the system may discard the previously recorded WAL logs. The checkpointing ensures that the modified data (dirty

[1]https://www.postgresql.org

[2]https://www.postgresql.org/docs/current/wal-configuration.html

pages) are already flushed to the disk (tables and indexes), preventing WAL from growing indefinitely. The checkpoint record stores PostgreSQL metadata about the current state of the database. This checkpointing process can be considered a combination of traditional checkpointing and log-based recovery.

Another database example is MySQL[3,4] 5.7 with the InnoDB Storage Engine. InnoDB employs a fuzzy checkpointing approach to flush dirty pages from the buffer pool to disk. Instead of flushing all dirty pages at once and degrading database performance, it flushes small batches of dirty pages. The checkpoint also represents the point in time when all cached data (dirty pages) have been flushed to disk.

3.3. Replication

Distributed systems rely on checkpoint/restore mechanisms to enhance fault tolerance for replicated systems, aiming to achieve higher throughput and basic fault tolerance by maintaining replicated states. As described in Section 3.2, logging, checkpointing, and recovery procedures are commonly adopted to implement fault-tolerant database systems. The most common approach is to augment passive replication, such as primary/backup, with durability strategies. This section emphasizes checkpoint-based recovery in active replication, which is typically implemented as State Machine Replicas.

Traditional State Machine Replication (SMR) models, as described in [Lamport 2019, Schneider 1990], may use checkpoints to save the application state in stable storage and recover a faulty replica by fetching the checkpoint state either from its own stable storage or from a checkpoint stored in a non-faulty replica.

Parallel State Machine Replication (PSMR) models, such as those presented in [Marandi et al. 2014], implement a multi-threaded replication model that takes advantage of modern hardware architectures. Given its parallelism, PSMR faces challenges in managing the system state to avoid the corruption that concurrent state changes could cause. As it needs to handle concurrent state changes, PSMR must also manage how checkpoints are saved, considering the concurrent modifications that may occur. For example, the work of [Mendizabal et al. 2014] proposes strategies for handling checkpointing in PSMR using either coordinated or uncoordinated checkpoints.

Another example of PSMR with checkpointing is seen in [Kotla and Dahlin 2004]. The study introduces a *parallelizer* service, called CBASE, to the PSMR model. This service routes commands to threads based on a partial order derived from the consensus agreement's batch of messages, allowing the system to execute commands in parallel. Application-specific rules and context determine the partial order, wherein the application understands the semantics of each command to decide if the command is independent and can be executed in parallel. This method circumvents the need for a total order after consensus is reached, enabling the parallel execution of independent commands. Total order refers to processing every command in the same order it was received. Checkpoints in CBASE are taken at fixed intervals, and every replica has the same checkpoint state generated. Checkpoint synchronization is achieved through the consensus algorithm among all replicas. The disadvantage is the overhead of this operation, which causes threads to wait until each of them finishes their work and completes the checkpoint.

[3]https://dev.mysql.com/

[4]https://dev.mysql.com/doc/refman/5.7/en/innodb-checkpoints.html

In [Kapritsos et al. 2012], a different approach to parallelize SMR is proposed, called Eve (Execution-Verify). This model features a primary replica that batches commands or requests and sends them to every replica. Each replica has a deterministic mixer that defines a subset of parallel-executable commands from each batch, likely producing the same results regardless of execution order. Although multiple batches may be in flight, they are executed sequentially, while subsets are executed in parallel. The model introduces a verification stage, where the output result of a batch is verified based on a hash of the final state. The verification process involves an agreement protocol determining whether the system needs to perform a rollback or can commit the commands and replay them to the client. If a rollback occurs, the replica replays the batch sequentially. The checkpoint process for Eve is similar to that of CBASE, but the periodic checkpoint is based on the number of executed batches instead of individual commands.

3.4. High Performance Computing

High-Performance Computing (HPC) is recognized as a strategic area to enable processing long-running tasks, such as those found in scientific computing. Among the problems solved in these environments is the simulation of physical and meteorological models, bioinformatics, recommendation systems, or artificial intelligence strategies that require platforms capable of processing large data and long execution.

Computing platforms for HPC are expensive and are typically available on demand, with access granted by reservation. Thus, once computing nodes' resources are in use, it is crucial to guarantee the success of the computation to avoid the need to restart long-term models from the beginning. In addition, as these are reservation-by-use environments, an interruption in the execution of a scientific model may not be resumed shortly. Returning to the allocation schedule is necessary to request the needed computational resources when they become available.

Checkpoint-based recovery can save the time of reprocessing a long-running computation. However, the additional computation resources used for durability would be used to increase the processing power. This dilemma between increasing performance or availability has been addressed in the literature.

Using probabilities, the study by [Frank et al. 2021] investigates checkpointing in the high-performance computing (HPC) context. It observes that most HPC jobs capable of checkpointing have only a medium probability of failure (MPF). The research suggests implementing checkpointing through an iterative algorithm that statistically calculates the optimal checkpoint intervals based on the probability of node crashes, given a newly proposed cost function. This approach identifies the number of checkpoints performed in an MPF job and models the average number of checkpoints in cases of failures while weighting the checkpointing cost based on the failure and success probabilities for a job. The study emphasizes that previous approaches have not considered that these jobs may not have a high probability of failure.

The study by [Tiwari et al. 2014] investigates the effects of traditional periodic checkpointing in extreme-scale systems, suggesting that it is possible to exploit the temporal locality of failures and to derive an approximation of the optimal checkpoint interval (OCI) instead of using a fixed interval. The study also proposes two novel algorithms for checkpointing: Lazy Checkpointing and Skip Checkpointing. The algorithms rely on

temporal locality to enhance the overall system computation performance by reducing unnecessary checkpoints and being more precise about when to take the checkpoint.

3.5. Cloud and Virtualized Environments

Checkpoint/restore strategies have been used in modern cloud and container environments allowing users and container management systems to migrate applications to another place. User-level checkpointing can be used to migrate containers from one physical machine to another, but this may cause a delay in the response time of the source container. In [Stoyanov and Kollingbaum 2018], authors address this issue and propose an efficient approach for live migration of Linux containers as a Checkpoint/Restore In Userspace[5] (CRIU) feature. The CRIU library is a notable and popular user-space checkpointing library that is still being updated today. It can checkpoint and restore any application in the Linux operating system as long as the kernel provides the required interfaces.

Docker[6], a widely used container engine, incorporates an experimental feature that utilizes CRIU for checkpointing and restoring. This feature enables the freezing of a running container by creating a checkpoint that can be later employed to restore the container. It helps accelerate applications with lengthy startup times, rewinding a process to an earlier point in time by restoring the application state from a checkpoint and migrating the container between machines.

Container migration also has an application for Data Centers. The transferring of stateless containers is a regular task. However, the transference for stateful containers may result in momentary service unavailability [Xu et al. 2020]. Sledge is a tool whose goal is to reduce the period of unavailability in migrating containers among cloud provider nodes. A primary controller is responsible for storing migration logs and coordinating migrations. For the migration, three elements should be transferred from one node to another: i) The actual real-time status, which is formed mainly by memory in use; ii) the writing/reading layers of the container; iii) the volume. An incremental checkpoint strategy provided by the CRIU minimizes the unavailability time to save the real-time status. Thus, only the portions of memory with updates after the last checkpoint need to be saved. Writing/reading layers usually have few modifications during execution and can be migrated using remote file synchronization tools such as the *rsync*. Finally, the volume is unmounted from the source container, CRIU restores the checkpoint files, and the volume is mounted in the new container. One problem is that the *Docker Daemon* reads the files of the containers stored during startup, and then the reboot is time-consuming.

More recently, Müller *at al.* [Müller et al. 2022] propose a sidecar that enables stateful containers to achieve fault tolerance. The work focuses on alleviating the challenges of checkpoint management by seamlessly integrating a Checkpoint/Restore (C/R) service into container orchestration systems. The checkpointing service, acting as an intermediary (sidecar), is tasked with creating snapshots of application containers and coordinating the checkpoint process alongside the request handling. Should any faults occur, a fresh application container is automatically generated from the most recent snapshot, allowing the intermediary to resume processing client requests from that specific checkpoint.

[5]http://criu.org/

[6]https://docs.docker.com/

In [Munhoz et al. 2022], authors demonstrate that checkpoint-based recovery can increase the availability when running unreliable spot machines. Spot machines are transient instances that cloud providers can revoke at any time without advance notice. In such environments, users are responsible for preemptively saving data and recovering applications. Their approach addresses the ephemeral spot instances as they are considerably cheaper resources and implement checkpointing/restore deployment strategies. The strategies comprise in-memory rollback restart at the MPI application level through User-Level Failure Mitigation (ULFM) and process-level rollback restart with the Berkeley Lab Checkpoint/Restart (BLCR) software package. Results indicate that the ULFM-based approach has the best performance and lowest cloud infrastructure cost, although it comes with a complex migration process. On the other hand, the BLCR strategy is based on a well-known software package and has a straightforward migration process, being a reasonable option for the rapid migration of legacy HPC applications to public clouds.

Checkpointing is vital for ensuring VMs' high availability, reliability, and fault tolerance in contemporary computing. By periodically capturing a VM's complete state, including memory, CPU, and disk, checkpoints serve as recovery references during system failures. This technique enables VM restoration, minimizing data loss and downtime. The study of [Gerofi and Ishikawa 2011] investigates the effects of dynamically adjusting checkpoint periods on performance degradation caused by checkpointing in VMs. Likewise, [Cully et al. 2008] introduced a service called Remus, which asynchronously transmits VM state changes to a backup host. In the event of a physical failure, Remus enables seamless operation on the alternate host, ensuring minimal downtime and preserving active network connections.

4. Final Remarks

This paper reviews more than 50 years of research and development of strategies for checkpointing. Despite being a well-known and consolidated topic, important advances in state-saving techniques and projects have emerged in recent years. Technological advances and cost reductions with hardware and management of computational infrastructures also allowed a wide adoption of these strategies in different applications in distributed computing.

The paper discusses approaches to speed up state saving, such as parallelism, partitioning, and copy-on-write, in addition to discussing the different strategies in coordination among processes to ensure safe and recoverable state saving. It is noted, in this sense, that uncoordinated approaches and the use of fuzzy checkpoints reduce durability costs during normal execution with the disadvantage of requiring a more complex recovery procedure. The frequency with which checkpoints are saved also directly impacts the cost. Whenever possible, avoiding state saving without compromising the reliability of the service is desirable. In this sense, studies have observed that application profile usage and workloads are useful to adjust the instants of activation of checkpoints.

Among the potential uses for checkpoints, in addition to their use in databases and the generation of global states, checkpoint-based recovery emerges as a viable alternative in HPC and cloud computing environments. In these models, guaranteeing tasks will be finished within the allocated time and keeping the infrastructure allocation cost low are

priority requirements. In this sense, checkpoints allow long computations to be resumed from more advanced points in case of failures.

Despite presenting an array of diverse checkpointing techniques, it is important to note that each context presents its own unique set of challenges, encompassing novel computational issues. In big data, managing vast datasets' size and storage is crucial. Costly computational power is required, but minimizing lost computation and ensuring fast recovery can save expenses For example, in the study by [Yan et al. 2016], the authors examine the application of checkpointing to big data tools, with the goal of reducing the number of cascaded re-computations resulting from evictions in unstable machines. Similarly, deep learning techniques also entail high training costs for models, requiring computational power over extended periods. The loss of a machine learning model's training state is undesirable. Consequently, recent research has proposed methods to implement enhanced checkpointing in this domain [Xing et al. 2015, Abadi et al. 2016, Nicolae et al. 2020]. Cloud computing represents another domain where recent studies have investigated the use of checkpointing to boost reliability. Data centers house a vast number of machines, increasing the likelihood of failures. Enhancing reliability in such environments is crucial, and recent research has addressed this issue [Zhao et al. 2017, Zhou et al. 2017]. Moreover, the growing popularity of containerized applications has given rise to the need for checkpointing to better manage container states [Chen 2015, Oh and Kim 2018, Müller et al. 2022].

With this paper, we gave an overview of the main checkpointing/restore techniques found in the literature and discussed their uses. We intend to contribute as a basis for researchers and educators in the area and point out some trends for the advancement of research and development in the area.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283.

Bessani, A., Santos, M., Felix, J., Neves, N., and Correia, M. (2013). On the Efficiency of durable state machine replication. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 169–180, San Jose, CA. USENIX Association.

Castro, M. and Liskov, B. (1999). Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA. USENIX Association.

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75.

Chandy, K. M. and Ramamoorthy, C. V. (1972). Rollback and recovery strategies for computer programs. *IEEE Transactions on computers*, 100(6):546–556.

Chen, Y. (2015). Checkpoint and restore of micro-service in docker containers. In *2015 3rd International Conference on Mechatronics and Industrial Informatics (ICMII 2015)*, pages 915–918. Atlantis Press.

Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. (2008). Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, page 161–174, USA. USENIX Association.

Duell, J. (2005). The design and implementation of berkeley lab's linux checkpoint/restart. Technical report, Lawrence Berkeley National Laboratory.

Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65:1302–1326.

Elmore, A. J., Das, S., Agrawal, D., and El Abbadi, A. (2011). Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 301–312.

Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408.

Elnozahy, E. N., Johnson, D. B., and Zwaenepoel, W. (1992). Measured performance of consistent checkpointing. In *Proceedings of the Eleventh Symposium on Reliable Distributed Systems*, number CONF.

Frank, A., Baumgartner, M., Salkhordeh, R., and Brinkmann, A. (2021). Improving checkpointing intervals by considering individual job failure probabilities. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 299–309.

Gerofi, B. and Ishikawa, Y. (2011). Workload adaptive checkpoint scheduling of virtual machine replication. In *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*, pages 204–213.

Janakiraman, G. and Tamir, Y. (1994). Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In *Proceedings of IEEE 13th Symposium on Reliable Distributed Systems*, pages 42–51. IEEE.

Junior, E. d. A. G. (2020). Redução do custo da durabilidade em replicação máquina de estados através de checkpoints particionados. Master's thesis, Universidade Federal do Rio Grande.

Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. (2012). All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 237–250, USA. USENIX Association.

Kotla, R. and Dahlin, M. (2004). High throughput byzantine fault tolerance. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, page 575, USA. IEEE Computer Society.

Lamport, L. (2019). Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196.

Leu, P.-J. and Bhargava, B. (1988). Concurrent robust checkpointing and recovery in distributed systems. In *Fourth International Conference on Data Engineering*, pages 154–155. IEEE Computer Society.

Marandi, P. J., Bezerra, C. E., and Pedone, F. (2014). Rethinking state-machine replication for parallelism. In *2014 IEEE 34th International Conference on Distributed Computing Systems*, pages 368–377. IEEE.

McGee, W. C. (1977). The informatiom management system ims/vs: Part ii: Data base facilities. *IBM Syst. J.*, 16(2):96–122.

Mendizabal, O. M., Dotti, F. L., and Pedone, F. (2016). Analysis of checkpointing overhead in parallel state machine replication. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 534–537.

Mendizabal, O. M., Dotti, F. L., and Pedone, F. (2017). High performance recovery for parallel state machine replication. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 34–44.

Mendizabal, O. M., Jalili Marandi, P., Dotti, F. L., and Pedone, F. (2014). Checkpointing in parallel state-machine replication. In *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18*, pages 123–138. Springer.

Mostefaoui, A. and Raynal, M. (1996). Efficient message logging for uncoordinated checkpointing protocols. In *Dependable Computing—EDCC-2: Second European Dependable Computing Conference Taormina, Italy, October 2–4, 1996 Proceedings 2*, pages 353–364. Springer.

Müller, R. H., Meinhardt, C., and Mendizabal, O. M. (2022). An architecture proposal for checkpoint/restore on stateful containers. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 267–270.

Munhoz, V., Castro, M., and Mendizabal, O. (2022). Strategies for fault-tolerant tightly-coupled hpc workloads running on low-budget spot cloud infrastructures. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 263–272.

Nicolae, B., Li, J., Wozniak, J. M., Bosilca, G., Dorier, M., and Cappello, F. (2020). Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 172–181.

Oh, S. and Kim, J. (2018). Stateful container migration employing checkpoint-based restoration for orchestrated container clusters. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 25–30.

Oppenheimer, G. and Clancy, K. (1968). Considerations for software protection and recovery from hardware failures in a multiaccess, multiprogramming, single processor system. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 29–37.

Plank, J. S., Beck, M., Kingsley, G., and Li, K. (1995). Libckpt: Transparent checkpointing under UNIX. In *USENIX 1995 Technical Conference*. USENIX Association.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319.

Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., and Verissimo, P. (2009). Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465.

Stoyanov, R. and Kollingbaum, M. J. (2018). Efficient live migration of linux containers. In *High Performance Computing: ISC High Performance 2018 International Workshops, Frankfurt/Main, Germany, June 28, 2018, Revised Selected Papers 33*, pages 184–193. Springer.

Tamir, Y. and Sequin, C. H. (1984). Error recovery in multicomputers using global checkpoints. In *13th International Conference on Parallel Processing*, pages 32–41.

Tiwari, D., Gupta, S., and Vazhkudai, S. S. (2014). Lazy checkpointing: Exploiting temporal locality in failures to mitigate checkpointing overheads on extreme-scale systems. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 25–36. IEEE.

Xing, E. P., Ho, Q., Dai, W., Kim, J. K., Wei, J., Lee, S., Zheng, X., Xie, P., Kumar, A., and Yu, Y. (2015). Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data*, 1(2):49–67.

Xu, B., Wu, S., Xiao, J., Jin, H., Zhang, Y., Shi, G., Lin, T., Rao, J., Yi, L., and Jiang, J. (2020). Sledge: Towards Efficient Live Migration of Docker Containers. *IEEE International Conference on Cloud Computing, CLOUD*, 2020-Octob:321–328.

Yan, Y., Gao, Y., Chen, Y., Guo, Z., Chen, B., and Moscibroda, T. (2016). Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 484–496, New York, NY, USA. Association for Computing Machinery.

Zhao, J., Xiang, Y., Lan, T., Huang, H. H., and Subramaniam, S. (2017). Elastic reliability optimization through peer-to-peer checkpointing in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 28(2):491–502.

Zheng, W., Tu, S., Kohler, E., and Liskov, B. (2014). Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 465–477, USA. USENIX Association.

Zhong, H. and Nieh, J. (2001). Crak: Linux checkpoint/restart as a kernel module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University.

Zhou, A., Sun, Q., and Li, J. (2017). Enhancing reliability via checkpointing in cloud computing systems. *China Communications*, 14(7):1–10.