Generic Checkpointing Support for Stream-based State-Machine Replication

Laura Lawniczak, Marco Ammon, and Tobias Distler
{lawniczak,marco.ammon,distler}@cs.fau.de
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Germany

Abstract

Stream-based replication facilitates the deployment and operation of state-machine replication protocols by running them as applications on top of data-stream processing frameworks. Taking advantage of platform-provided features, this approach makes it possible to significantly minimize implementation complexity at the protocol level. To further extend the associated benefits, in this paper we examine how the concept can be used to provide generic support for creating, storing, and applying checkpoints of replica states, both in the use case for catch up and garbage collection as well as to recover failed replicas. Specifically, we present three checkpointing-mechanism designs with different degrees of platform involvement and evaluate them in the context of Twitter's stream-processing engine Heron.

CCS Concepts: • **Computer systems organization → Reliability**.

Keywords: State-machine replication, checkpointing, recovery, data-stream processing

ACM Reference Format:
Laura Lawniczak, Marco Ammon, and Tobias Distler. 2023. Generic Checkpointing Support for Stream-based State-Machine Replication. In *10th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '23), May 8, 2023, Rome, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3578358.3591329

1 Introduction

Implementing and operating replicated systems is an inherently difficult task due to the high complexity associated with their underlying protocols (e.g., Paxos [14], Raft [19]). Stream-based state-machine replication [15] mitigates this problem by designing protocols in such a way that they can run as applications on data-stream processing frameworks like Heron [13] or Storm [21]. With these platforms already offering support for essential tasks such as the deployment of code, the creation and management of network connections, as well as the recovery of failed processing-node instances, as a key benefit, this approach leads to significantly smaller and less complex protocol implementations, as recently demonstrated by the crash-tolerant Tara protocol [15].

When developing a stream-based protocol, in most cases it is straightforward to determine how to leverage an existing platform-provided feature, because in general there essentially is only one way to integrate the corresponding functionality. However, this specifically does not apply to the problem of designing an efficient checkpoint mechanism, which is crucial to allow trailing replicas to update their states and to enable failed replicas to recover from crashes. As we show in this paper, with respect to checkpointing there are multiple design alternatives, especially when pursuing the goal of exploiting the particular characteristics of stream-based protocol architectures for efficiency.

To explore the design space, we present and evaluate multiple generic checkpointing mechanisms with varying degrees of platform involvement, ranging from a purely application-based solution (in which the protocol itself is responsible for capturing, storing, and transferring checkpoint data) to a highly platform-based approach (in which the protocol only needs to capture and write back checkpoint data when instructed to do so by the underlying framework). Apart from the checkpoint use cases commonly supported by traditional replication-protocol implementations such as BFT-SMaRt [3] (i.e., enabling slow or newly started replicas to catch up), we are specifically interested in examining an additional feature closely related to checkpointing: the automatic recovery of a crashed replica instance, possibly on another server in case the entire physical machine failed. Requiring access to the surrounding computing infrastructure, such recovery mechanisms typically are not part of replication libraries [3, 12]. In contrast, with data-stream processing framework like Heron (assisted by its resource manager Mesos) controlling the entire cluster environment, support for automatic recovery is readily available to stream-based replication protocols.

PaPoC '23, May 8, 2023, Rome, Italy
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *10th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '23), May 8, 2023, Rome, Italy*, https://doi.org/10.1145/3578358.3591329.

In summary, this paper makes the following contributions: (1) It shows that in stream-based replication protocols such as Tara different stages have different requirements with regard to checkpointing. (2) It presents three designs for offering checkpoint support to stream-based protocols, including both application-based and platform-based approaches. (3) It experimentally evaluates the proposed techniques in the context of a fault-tolerant coordination service.

The remainder of this paper is structured as follows: Section 2 provides background on stream-based replication in general, and the Tara protocol in particular. Section 3 identifies the requirements checkpointing mechanisms need to fulfill for stream-based replication. Section 4 details our three designs and elaborates on possible implementation variants. Section 5 evaluates the proposed designs during normal-case operation and in the presence of node crashes. Finally, Section 6 discusses related work and Section 7 concludes.

2 Background

State-machine replication protocols allow a system to tolerate failures by composing the system as a group of replicated servers all maintaining an instance of the application state. Stream-based state-machine replication [15] facilitates the implementation and deployment of such a system by running the replication protocol as an application on top of a stream-processing framework. For this purpose, the protocol is designed as a set of stages that are organized in a directed acyclic graph through which information flows in the form of data tuples. Figure 1 illustrates this concept using the five main stages of the Tara protocol [15], which are responsible for the agreement and execution of client commands.

2.1 Replication Protocol

Tara's main replication process consists of the following steps (see Figure 1). When new client commands arrive, the underlying framework inserts them into one of its input queues from where they enter the protocol via the request-source stage. Next, commands flow through a sequence of three stages (i.e., proposers, committers, and executors) that together implement a crash-tolerant consensus algorithm responsible for assigning a unique sequence number to each command. Once this assignment is confirmed, each executor processes the commands in the order of their sequence numbers and forwards the corresponding results to the reply-sink stage, where they are eventually delivered to clients.

To offer the same fault tolerance properties as traditional state-machine replication protocols, each Tara stage comprises a group of replicas. A replica is represented as an individual processing-node instance in the protocol's topology and together they form a fully replicated system. The number of replicas in each stage depends on the specific responsibilities and varies between $f+1$ (for request sources, proposers, and reply sinks) and $2f+1$ (for committers and executors) to tolerate up to f crashes per stage. Of the $f+1$ proposers only a single one, the leader, actively participates in the protocol. If this replica fails, a view-change mechanism ensures that the leader role is assigned to one of the remaining proposers.

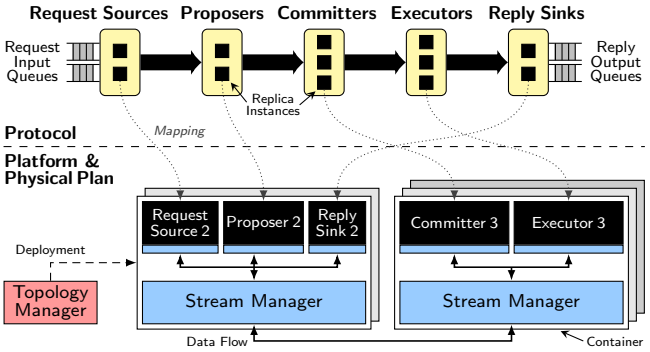


Figure 1. Stream-based state-machine replication

2.2 Execution Platform

Stream-processing frameworks execute applications as a series of processing nodes organized in a directed acyclic graph. With the topology of a stream-based replication protocol being structured in such a way, it can be executed as an application on platforms such as Twitter's stream-processing framework Heron [13]. At startup, the protocol topology is translated into a physical execution plan defining how the nodes are distributed among servers. As shown in Figure 1, this step allows replicas of different stages to be packed into the same container. Containers are Heron's unit of deployment and each of them is run by a local *stream manager*, which, for example, takes care of routing data tuples (within its own container and between containers), thereby enabling replicas to communicate.

At the global level, a central *topology manager* is responsible for handling tasks that affect the entire system. This includes the deployment and configuration of containers during startup as well as (assisted by stream managers) the monitoring and recovery of processing nodes at runtime. Notice that once deployed, a stream-based protocol is able to run on its own even if the topology manager becomes (temporarily) unavailable: an active participation of the topology manager is only necessary to recover replicas after faults. This is thanks to (1) the topology manager not being directly involved in the data flow and (2) the protocol's built-in fault-tolerance and view-change mechanisms operating at the application level. By itself, the protocol can tolerate up to f concurrent replica failures per stage. Notice that it is important to configure the topology and container distribution in such a way that each physical server only hosts at most one node of each stage, ensuring independent replicas.

3 Stage-Specific Checkpointing

Like traditional replication protocols [14, 19], stream-based protocols rely on checkpoints (i.e., snapshots of replica states) to allow trailing replicas to catch up and to recover crashed replicas. In this section, we show that in stream-based architectures, however, different stages have both different characteristics and use cases with regard to checkpointing. In particular, we discovered that stages can be classified into two categories requiring *group checkpoints* and *instance checkpoints*, respectively (see Table 1). Such categorization is useful as it allows us to develop tailored checkpointing mechanisms in Section 4.

3.1 Group Checkpoints

This type of checkpoint represents the traditional notion of checkpointing in state-machine replication protocols and is used in stages for which it is ensured that all correct replicas eventually are in an identical state after having processed the same inputs. In general, this is true for later protocol stages that operate on the sequence of commands produced by the consensus process, as it is for example the case for the executors in Tara. With all replicas possessing consistent states, the snapshot created by one replica can **bring another replica of the same group up to speed**, which is why we refer to this category as group checkpoints.

To minimize overhead, group checkpoints are created periodically, for example every 1,000th sequence number. A larger checkpoint interval (i.e., less frequent snapshots) can prolong recovery, but on the other hand usually improves the performance in fault-free periods, especially for replicas with a large state to capture. For group checkpoints, the size of the replica state is typically dominated by the state of the replicated service and therefore varies between use cases.

Similar to other state-machine replication protocols [14, 19], group checkpoints in the context of stream-based replication are an intrinsic part of the protocol since they enable replicas to garbage-collect information on earlier sequence numbers [15]. Consequently, creation and application of such checkpoints should still be possible even if parts of the underlying stream-processing framework, and the topology manager in particular, are currently not available.

3.2 Instance Checkpoints

This kind of checkpoint is used by stages in which correct replicas may observe different inputs and thus end up in different states. A textbook example for such a stage are Tara's committers, which are responsible for replicating the leader's proposals across multiple servers. Specifically, when the acting leader crashes, a subset of committers may receive proposals that due to the failure do not reach all committers. To ensure correctness, it is essential that in the subsequent view change each committer precisely reports the proposals it has acknowledged in the past and is not allowed to lose any externalized information. This means that it is not feasible to share snapshots among replicas, as done with group checkpoints. Instead, these stages require checkpoints that **enable the recovery of a specific replica instance** (e.g., Committer 3, not simply any committer).

Recovering a specific replica instance makes it necessary to restore all state updates that were externalized via the replica's outputs prior to the crash and therefore might have been observed by others. Although the state of the affected stages is typically small and (due to belonging to a generic consensus algorithm) does not depend on the size of the replicated service, creating a new instance checkpoint for each output tuple usually results in significant overhead. This problem can be mitigated by persisting multiple subsequent state updates in a single checkpoint, similar to the batching of commits in log-based file systems and databases [9].

In contrast to group checkpoints, instance checkpoints are not an integral part of the stream-based replication protocol but instead a means for the stream-processing platform to recover individual replicas, possibly on a different server. With the recovery from an instance checkpoint being initiated and performed by the platform, there is an inherent dependency between the checkpointing mechanism for replica instances and the platform component responsible for their recovery. In particular, this means that instance checkpoints only need to be accessible when the topology manager is available.

Table 1. Comparison of group and instance checkpoints

	Group Checkpoints	Instance Checkpoints
Applicability	Each replica of the same group	Specific replica instance
Granularity	Sequence-number interval	Each externalized update
Size	Use-case specific	Small
Recovery	Protocol level	Platform level

3.3 General Requirements

To limit the design space, in this paper we focus on checkpointing mechanisms that (independent of the specific checkpoint type) store a replica's state in a synchronous manner, meaning that the snapshot-creation procedure must block until all captured data has reached its destination (e.g., local storage, another server); exploiting asynchronous approaches [7] is part of future work. Once a checkpoint has been created, older checkpoints by the same replica may be garbage-collected. That is, for each replica it is sufficient to only keep the latest group/instance checkpoint.

4 Checkpointing Techniques

In this section, we introduce three checkpointing-mechanism designs for stream-based replication that represent different tradeoffs with regard to the degree of platform involvement. As summarized in Table 2, two of the approaches are tailored to a specific type of checkpoint, whereas the

Table 2. Overview of the checkpointing techniques analyzed in this paper and the types of checkpoints (CPs) they support.

	Group CPs	Instance CPs
Protocol-driven checkpointing	Yes	No
Global platform-based checkpointing	No	Yes
Local platform-based checkpointing	Yes	Yes

third is able to support both group checkpoints as well as instance checkpoints. In the following, we discuss each of the three techniques in detail. When it comes to implementation-related specifics, we focus on Heron as underlying stream-processing framework since this is the platform we use in our prototype. However, the general concepts are also adaptable to other stream-processing frameworks.

4.1 Protocol-Driven Checkpointing

The main idea behind our first technique, *protocol-driven checkpointing (PDC)*, is to delegate full responsibility for checkpoint creation and management to the stream-based replication protocol. Specifically, this means that each replica itself is equipped with procedures and resources for deciding when to create a new checkpoint, storing its latest snapshot, and transferring a checkpoint to another replica on demand.

Providing the entire checkpointing functionality at the application level, PDC has the benefit of achieving platform independence, but on the other hand adds complexity to the replication-protocol implementation. Furthermore, with persistent storage being managed by the underlying framework and therefore not available to this approach, each replica needs to keep its latest snapshot in memory. As a consequence, in order to transfer a checkpoint between two replicas, both of the replicas must be running. This prevents PDC from being used for instance checkpoints, whose purpose is to recover the same replica instance after a crash. With regard to group checkpoints, on the other hand, this does not pose a problem. Here, the size of the replica group (e.g., $2f + 1$ executors in Tara) ensures that there always is at least one correct replica that has the latest checkpoint and can provide it to trailing or recovering replicas.

4.2 Global Platform-based Checkpointing

The key design goal of our second technique, *global platform-based checkpointing (GPC)*, is to leverage mechanisms that already exist in stream-processing frameworks to enable the recovery of stateful topologies. As a result, GPC requires only limited implementation work at the protocol level, which mainly consists of routines to retrieve a replica's internal state and to update its state based on a provided snapshot. Due to the checkpointing process being initiated by the topology manager, GPC does not support group checkpoints; as discussed in Section 3, group checkpoints need to be producible even in the topology manager's absence and replicas must be able to request checkpoints themselves.

In Heron, the recovery of stateful stream-based applications is based on a coordinated rollback-recovery scheme [8], which ensures that all processing nodes (i.e., in our case: replicas) in the system create consistent checkpoints. As illustrated in Figure 2, the platform for this purpose inserts special markers into the stream of regular data tuples. Starting at the roots of the directed acyclic graph, the markers are forwarded from stage to stage, thereby eventually reaching all replicas. Whenever a replica receives a marker M_i for a new checkpoint number i, the replica temporarily suspends the processing of data tuples from the input stream through which the marker arrived. Once the replica has seen the same marker in all of its input streams, the replica creates a snapshot of its state, adds the marker M_i to all of its output streams, and resumes processing data tuples. With all replicas behaving accordingly, this approach ensures that their checkpoints together represent a consistent snapshot of the system state at a specific point in time (i.e., a specific sequence number).

Apart from capturing replica states, all data tuples transmitted since the latest global checkpoint are also recorded in Heron. In combination, this allows the framework to recover a protocol topology by first restarting all replicas, then applying the respective snapshots, and issuing the redelivery of logged data tuples in the recorded order. Always restoring the protocol in its entirety, this approach is primarily suitable for scenarios in which a larger number of replicas (typically belonging to multiple stages) need to be recovered at once.

Using GPC for instance checkpoints makes it necessary to ensure that each replica's outputs are only forwarded to the next stage after the corresponding state updates have been stored in a checkpoint (see Section 3.2). With Heron's built-in mechanism only initiating checkpoints at the granularity of seconds, it is advisable to combine all of a replica's state updates since the latest checkpoint and persist them together in a single batch. Even though this does not eliminate the need to delay the externalization of outputs (which is required for correctness), this has the benefit of reducing the number of costly write operations to stable storage.

4.3 Local Platform-based Checkpointing

To avoid the drawbacks associated with GPC while still exploiting framework-provided checkpointing features, our third technique, *local platform-based checkpointing (LPC)*, addresses checkpointing at the replica level (in contrast to the

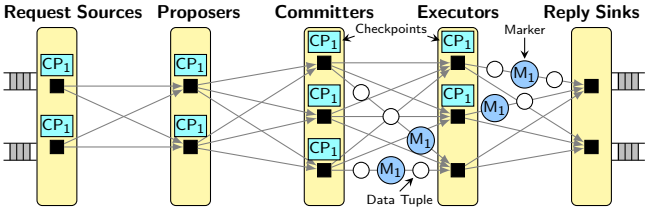
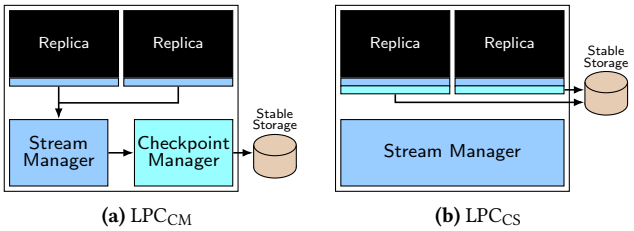


Figure 2. Distributed checkpointing in GPC



(a) LPC_{CM} **(b)** LPC_{CS}

Figure 3. Persisting a checkpoint in the two LPC variants

topology level as in GPC). Using LPC, replicas themselves decide when to trigger the creation of a new checkpoint; all following steps are then handled by replica-local platform components. Without the topology manager being involved in snapshot generation, this approach enables LPC to support both group checkpoints and instance checkpoints.

Starting from the general LPC concept, we designed two specific variants that differ in the way the snapshot data is persisted to stable storage: (1) As shown in Figure 3a, LPC_{CM} for this purpose follows Heron's built-in path that entails replicas transmitting the checkpoint contents to their local stream manager. From there, the data is forwarded to a dedicated checkpoint manager (CM), which is responsible for performing the actual write to stable storage. (2) In contrast, in LPC_{CS} replicas directly interact with the stable storage through a lightweight protocol-independent checkpointing shim (CS) that is added to the execution environment that replicas run in (see Figure 3b). Compared with LPC_{CM}, this offers the benefit of significantly reducing the inter-process communication necessary to store (and load) a checkpoint.

Independent of the particular variant, LPC supports the use of batched writes to improve efficiency. Different from GPC, batching in LPC however is not time-based but instead applied at the granularity of state updates. As a key advantage, this makes it possible to produce checkpoints more frequently and thereby minimize the latency overhead induced by instance checkpoints (see Section 5.2).

5 Evaluation

This section evaluates the three presented checkpointing techniques (with both variants of LPC) using a coordination service that is replicated by Tara as an application on Heron. We use a cluster of three servers for the replicas and one server acting as the Heron main node that among other things hosts the topology manager (all Intel Xeon CPU E3-1245 v3, 3,40 GHz, 8GB RAM). All platform-based checkpoints use the file-system interface provided by Heron, storing the data with NFS on a dedicated server with an SSD. The servers are connected with 1 Gbit Ethernet. Clients run on up to three additional machines and submit requests in a closed loop. Each result represents the average of three runs.

5.1 Group-Checkpoint Performance

In our first experiment, we evaluate the performance impact of managing group checkpoints for the two of our techniques that support this type of checkpoint: PDC and LPC. We checkpoint executors as this is the only stage in Tara that uses group checkpoints. The system is configured to store a checkpoint every 1,000 requests. As can be seen in Figure 4, it is possible to use platform-based checkpointing without impacting overall system performance. Both PDC and LPC_{CM} show a nearly identical performance with a maximum throughput of about 13,000 requests. While LPC_{CS} still shows an adequate performance, there is a decline in both throughput and latency of up to 20% under high load. This is likely due to the fact that LPC_{CS} does not shift the load of managing asynchronous tasks (e.g., the deletion of old checkpoints) to the checkpoint manager, thus having more impact on the performance of the actual replicas in comparison with LPC_{CM}.

5.2 Instance-Checkpoint Performance

In our next experiment, we inspect the performance of the system under full crash-recovery conditions, storing instance checkpoints for each node in a way that it can be safely reintroduced into the system after a failure. Of course, stronger resilience properties achieved by synchronous logging do not come without cost [2], which is why performance in this experiment is substantially lower than in the previous one.

The results in Figure 5 show a distinct difference between GPC and the two LPC variants. Since GPC triggers checkpoints in a fixed time interval and to avoid publishing potentially lost data, messages are withheld until they are saved in a checkpoint. This introduces a high latency even under light load since each request has to wait on the next checkpoint in at least three stages. With a configured checkpoint interval of 500ms this results in an average latency of around 1.2s. Originally, Heron only offers a minimal interval of 1s. We removed this configuration restriction, but below 500ms the system was overwhelmed by the number of checkpoints. However, this time-based checkpointing also has the advantage of allowing the system to adapt to the load, enabling a larger throughput with a high but still stable latency.

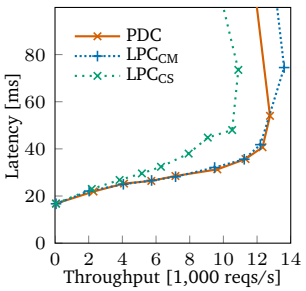


Figure 4. Throughput vs. latency: group checkpoints

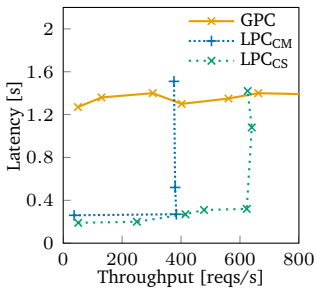


Figure 5. Throughput vs. latency: instance checkpoints

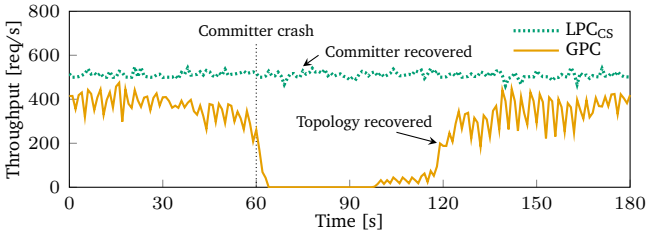


Figure 6. Impact of non-leader replica failure

LPC does not have this restriction, but for every request needs to save a checkpoint at multiple nodes. Hence, we introduce batching in both the request source and committer stage to mitigate this. With a batch factor of up to 10 in both stages, LPC$_{CM}$ has a comparably low latency of approximately 300ms and can reach a maximum throughput of around 400 requests per second. A close inspection shows that with multiple replicas persisting checkpoints in a high frequency, the single checkpoint manager on each server quickly becomes the bottleneck. Since LPC$_{CS}$ shares this load amongst its replicas, it can reach a higher throughput of 600 requests per second with the same latency.

5.3 Node-Crash Impact

As was shown in the previous experiment, both methods allowing full crash recovery have their advantages, with GPC offering a higher and more adaptive throughput while LPC being able to provide lower latency, especially for low to average load. Hence, as an additional decision criteria, we now evaluate the impact of a single node crash on the system. For LPC, we only consider LPC$_{CS}$ since it offers a better performance under crash-recovery conditions than LPC$_{CM}$.

In our experiment shown in Figure 6, we deliberately crash one committer replica after 60 seconds of service. Using LPC$_{CS}$, the crash (and subsequent automatic recovery) of a single non-proposer replica has no impact on system throughput. For GPC, on the other hand, one sees the vast impact of the global recovery scheme. As even the crash of a single node triggers the rollback and recovery of the whole topology, the system experiences a significant downtime up to a minute. In regards to availability during and after a replica crash, our experiments confirm that a local recovery technique has a huge advantage over the existing global mechanism offered by Heron.

6 Related Work

Checkpointing in state-machine replication is a crucial topic, which is why there has been a lot of research in the area of both the creation of checkpoints [4, 7, 18, 20] and their impact on performance [2, 17]. In this paper, we focus not on the creation but the management of checkpoints, especially in combination with utilizing the infrastructure provided by the stream-processing platform. Hence, these insights into checkpoint creation are complementary to our work.

Table 3. Summary of the analyzed checkpointing techniques

	Throughput	Scalability	Latency Overhead	Crash-Recovery	Complexity Overhead
PDC	+	○	+	n.a.	−
GPC	○	+	−	−	○
LPC	−	−	○	+	+

Similar, there exist state-machine replication protocols that specifically discuss the recovery [2, 3, 5, 6, 11, 12] of replicas, both of the same instance as well as on another server in combination with reconfiguration [1]. For JPaxos, Kończak et al. [12] for example developed different recovery algorithms that rely on the protocol's snapshot mechanism for catch-up and recovery. These approaches have in common that the mechanisms are handled on a protocol level, comparable to PDC. To our knowledge, there has not been a study on outsourcing these tasks to an underlying platform.

One large factor in the performance and applicability of persistent storage for checkpoints is the underlying storage technique and hardware. In recent years, there has been much advancement in reducing the latency of network storage by using new techniques with NVM [10, 16, 22]. Combined with the proposed platform-based techniques GPC and LPC, this can help bridge the gap in performance compared to no logging or only storing checkpoints in volatile memory.

7 Conclusion

In this paper, we presented different techniques for handling checkpoints in stream-based protocols: (1) PDC being purely managed by the protocol itself, (2) GPC relying mostly on the existing checkpointing mechanism offered by the stream-processing platforms and working on a topology level, and (3) LPC (in two variants), a more instance focused approach, utilizing the platform's resources but managing the replicas individually, granting more control to the protocol itself.

Our analysis and the evaluation with Tara shows that all approaches have their advantages and drawbacks (see Table 3): PDC offers among the highest throughput and best latency, but does not provide any crash-recovery for most nodes. Additionally, it has a high complexity overhead in the protocol implementation as all checkpoint management needs to be done by the application itself. The time-triggered checkpointing of GPC allows the system to automatically scale with the load and provide a stable – if high – latency, but on a replica crash, the whole system experiences a massive downtime. Also, group checkpoints still need to be managed by the protocol itself. With LPC, both group and instance checkpoints can be handled by the stream-processing platform and the system is not affected by an individual node crash. However, the maximum throughput is bound by the configuration of optimizations such as batching.

Acknowledgments: Partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 446811880.

References

[1] Eduardo Alchieri, Fernando Dotti, Odorico M Mendizabal, and Fernando Pedone. 2017. Reconfiguring Parallel State Machine Replication. In *Proceedings of the 36th International Symposium on Reliable Distributed Systems (SRDS '17)*. 104–113.

[2] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. 2013. On the Efficiency of Durable State Machine Replication. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC '13)*. 169–180.

[3] Alysson Bessani, João Sousa, and Eduardo E P Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN '14)*. 355–362.

[4] Tobias Distler. 2021. Byzantine Fault-Tolerant State-Machine Replication from a Systems Perspective. *Comput. Surveys* 54, 1, Article 24 (2021), 38 pages.

[5] Tobias Distler, Rüdiger Kapitza, Ivan Popov, Hans P. Reiser, and Wolfgang Schröder-Preikschat. 2011. SPARE: Replicas on Hold. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS '11)*. 407–420.

[6] Tobias Distler, Rüdiger Kapitza, and Hans P. Reiser. 2010. State Transfer for Hypervisor-Based Proactive Recovery of Heterogeneous Replicated Services. In *Proceedings of the 5th "Sicherheit, Schutz und Zuverlässigkeit" Conference (SICHERHEIT '10)*. 61–72.

[7] Michael Eischer, Markus Büttner, and Tobias Distler. 2019. Deterministic Fuzzy Checkpoints. In *Proceedings of the 38th International Symposium on Reliable Distributed Systems (SRDS '19)*. 153–162.

[8] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *Comput. Surveys* 34, 3 (2002), 375–408.

[9] Robert Hagmann. 1987. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th Symposium on Operating Systems Principles (SOSP '87)*. 155–162.

[10] Jaehyun Hwang and Qizhe Cai. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. 127–140.

[11] Jan Kończak and Paweł T Wojciechowski. 2021. Failure Recovery from Persistent Memory in Paxos-Based State Machine Replication. In *Proceedings of the 40th International Symposium on Reliable Distributed Systems (SRDS '21)*. 88–98.

[12] Jan Kończak, Paweł T Wojciechowski, Nuno Santos, Tomasz Żurkowski, and André Schiper. 2019. Recovery Algorithms for Paxos-Based State Machine Replication. *IEEE Transactions on Dependable and Secure Computing* 18, 2 (2019), 623–640.

[13] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 41st International Conference on Management of Data (SIGMOD '15)*. 239–250.

[14] Leslie Lamport. 1998. The Part-time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[15] Laura Lawniczak and Tobias Distler. 2021. Stream-based State Machine Replication. In *Proceedings of the 17th European Dependable Computing Conference (EDCC '21)*. 119–126.

[16] Xiaojian Liao, Zhe Yang, and Jiwu Shu. 2022. RIO: Order-Preserving and CPU-Efficient Remote Storage Access. *arXiv preprint arXiv:2210.08934* (2022).

[17] Odorico M Mendizabal, Fernando Luís Dotti, and Fernando Pedone. 2016. Analysis of Checkpointing Overhead in Parallel State Machine Replication. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16)*. 534–537.

[18] Odorico M Mendizabal, Parisa Jalili Marandi, Fernando Luís Dotti, and Fernando Pedone. 2014. Checkpointing in Parallel State-Machine Replication. In *Proceedings of the 18th International Conference on Principles of Distributed Systems (OPODIS '14)*. 123–138.

[19] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*. 305–320.

[20] Tuanir F Rezende, Pierre Sutra, Rodrigo Q Saramago, and Lasaro Camargos. 2017. On Making Generalized Paxos Practical. In *Proceedings of the 31st International Conference on Advanced Information Networking and Applications (AINA '17)*. 347–354.

[21] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm @Twitter. In *Proceedings of the 40th International Conference on Management of Data (SIGMOD '14)*. 147–156.

[22] Qingfeng Zhuge, Hao Zhang, Edwin Hsing-Mean Sha, Rui Xu, Jun Liu, and Shengyu Zhang. 2021. Exploring Efficient Architectures on Remote In-Memory NVM over RDMA. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 5s (2021), 1–20.