SMaRtTrie: Reducing Checkpoint's Impact in SMR Systems with a CTrie Data Structure

Erick Pintor[1], **Fernando Luís Dotti**[1]

[1]Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre, RS – Brazil

erick.pintor@edu.pucrs.br, fernando.dotti@pucrs.br

Abstract. *State Machine Replication (SMR) is a well-established approach for the development of fault tolerant systems through replication. Durable SMR systems rely on command logging and checkpoints for fast recovery upon failures. However, checkpoints are known to impact service's throughput as traditional SMR architectures stop serving requests in order to ensure checkpoint consistency. In this paper, we present SMaRtTrie: a simple in-memory key-value storage built on a concurrent trie data structure that supports the creation of consistent checkpoints in parallel with command execution. Experimental evaluation shows that SMaRtTrie can reach the same throughput as other commonly used solution while sustaining* 78% *throughput during checkpoints.*

1. Introduction

State Machine Replication (SMR) [Lamport 1978, Schneider 1990] is a well-established approach for the development of fault-tolerant systems. In an SMR system, its state is replicated in multiple machines called replicas. Replicas receive commands from clients and determine their total order of execution via a consensus protocol. Commands are deterministically executed in the order agreed upon consensus. Starting from a same initial state, replicas apply the same sequence of commands and therefore traverse the same sequence of states. An SMR system can tolerate a number of faulty replicas without losing state consistency, making it an important technique to sustain high availability in large distributed systems.

It is often required that an SMR system is made durable so that it can survive the crash or shutdown of all its replicas without losing its state. Furthermore, upon a failed replica a replacement has to be introduced with the same state in order to maintain the availability level. Traditionally, durability in SMR systems is achieved by logging commands into durable storage in the consensus order. A recovering replica can rebuilt its state by re-executing all commands in the log. To improve resource utilization and recovery performance on large data sets, checkpoints of the in-memory state are stored periodically into a durable storage. In a checkpoint enabled system, a recovering replica can restore its state by loading the last checkpoint created and re-executing only commands logged after its creation.

While recovery is mandatory for most highly-available systems, and checkpoints are thus needed for practical use-cases, they are known to impact replica's throughput with collateral effects throughout the system. Traditional SMR applications stop executing commands while creating checkpoints in order to preserve their consistency, which effectively drops the replica's throughput to zero [Bessani et al. 2013] and might create

an unavailability window if occurring in a majority of replicas at the same time. Alternative approaches [Bessani et al. 2013, Zheng et al. 2014] include cross node coordination to prevent replicas taking checkpoints at the same time, and the usage of non-consistent snapshots that can be taken in parallel with command execution. However, both solutions add considerable overall complexity.

In "Concurrent tries with efficient non-blocking snapshots" [Prokopec et al. 2012] a concurrent trie is proposed (a.k.a.: CTrie) that allows for consistent snapshots to be taken in constant time ($\mathcal{O}(1)$). In this paper, we investigate the benefit of using this kind of data structure as the base for state management in an SMR system. We propose SMaRtTrie: a simple in-memory key-value store based on a concurrent trie. Our experiments show that SMaRtTrie reaches the same throughput without checkpoints as other commonly used data structure, and sustain 78% throughput during checkpoint execution (Fig. 5). Additionally, this paper presents an orthogonal discussion about the impact of garbage collection algorithms in checkpoint enabled applications on the underlying distribution platform (BFTSMaRt [Bessani et al. 2014]).

In Section 2 we present the underlying system model and assumptions. Section 3 presents background information on the systems and techniques discussed as well as a discussion on the checkpoint's performance impact in SMR applications along with related work. SMaRtTrie's implementation details are presented in Section 4 and its evaluation in Section 5. We conclude this paper by presenting final considerations in Section 6.

2. System Model and Assumptions

We assume a distributed system composed by interconnected processes: there is a bounded set $R = \{r_1, r_2, ...r_n\}$ with n replica processes and an unbounded set $C = \{c_1, c_2, ...\}$ of client processes. We assume the crash failure model, excluding arbitrary or malicious behavior. After crashing, a process may recover. A process is said correct if it eventually remains up forever, or faulty otherwise. To sustain agreement, and thus service availability, $n >= 2f + 1$ replicas are needed, being f the number of tolerated crashes. Violating this assumption could cause data loss ([Bessani et al. 2013]). Moreover, recovery should be possible in the event of all replicas being compromised. In such cases, to keep state consistency, we assume standard durability mechanisms.

Processes communicate exclusively by message passing, assuming fair-loss links: if a sender sends a message enough times it is eventually delivered to correct receivers. Communication may be both one-to-one and one-to-many. In the later, communication assumes atomic broadcast properties [Défago et al. 2004]. Additional synchrony assumptions for atomic broadcast [Fischer et al. 1985] are not discussed in this paper. We extend the atomic broadcast $deliver(m)$ primitive to $deliver(i, m)$, where m is the delivered message and i is the consensus instance, thus allowing to determine if checkpoints must take place and to calculate which messages should be retrieved upon failure recovery.

Our consistency criterion is linearizability [Herlihy and Wing 1990]. An execution is linearizable if it's possible to totally order the operations such that: it respects the semantics of the objects accessed by the operations as expressed in their sequential specifications; and it respects the real-time ordering of the operations in the execution. There is a real-time order among two operations if one operation finishes in a client before the other operation starts in the same client.

3. Background

To contextualize this paper's contributions, this section reviews the primary concepts involved: State Machine Replication (SMR) along with the problems that emerge from checkpoints in SMR systems; related work to mitigate checkpoint impact in SMR applications; and the concurrent trie data structure investigated.

3.1. State Machine Replication (SMR)

State Machine Replication [Lamport 1978, Schneider 1990] is an approach to build fault tolerant systems that achieves high availability through replication. In an SMR system, machines called replicas contain each a copy of the system's state. State access is made through commands submitted by clients to any replica in the system. Replicas determine the commands' total execution order by running an instance of a consensus protocol such as Paxos [Lamport et al. 2001] or Raft [Ongaro and Ousterhout 2014]. After executing a command, the replica acknowledges its completion back to the submitting client. A fundamental property ensures state consistency across replicas: given that all replicas start at the same initial state, and deterministically execute all commands in the same order, they all eventually reach the same final state. The system's replicated architecture allows it to tolerate a number of faulty replicas without consistency loss. Conservatively, the system can continue to execute client issued commands as long as a majority of replicas stay operational.

Durability is required to ensure the system can survive the crash or shutdown of all its replicas without losing its state. Traditionally, durability in SMR systems is achieved through command logging: after each round of consensus, the resulting commands are persisted in a durable storage in the order agreed upon. A recovering replica can restore its state by re-executing all commands in the log. It is worth noting that a recovering replica's log could become outdated if new commands are processed by the remaining non-recovering replicas. To reconcile, the recovery process must transfer and execute missing commands from other non-recovering replicas.

Command logging can consume large amounts of storage space in high throughput systems. Moreover, recovering a replica's state form a large log can be time-consuming as the system has to go through every state transition that ever occurred in order to restore the state prior to a crash or shutdown. To reduce storage space consumption and improve recovery performance, a technique called checkpoint is applied: replicas periodically persists a snapshot of their consolidated state into durable storage. After a checkpoint, commands logged prior to its creation can be removed from storage as their effects have been persisted by the newly created snapshot. In a checkpoint enabled system, a recovering replica can restore its state by first loading its last snapshot, then only re-executing commands logged after its creation.

3.2. Checkpoint's Performance Problems and Related Work

In order to create a consistent snapshot of the system's state, traditional SMR applications stop executing commands while creating a checkpoint. This solution reduces a replica's throughput to zero for the duration of the checkpoint process. Furthermore, if a majority of replicas start a checkpoint at approximately the same time, it creates a period of generalized unavailability. Adding hardware, such as fast durable storage, might speed up the

process of persisting a snapshot, however, it does not fundamentally solve the problem which is an artifact of the established algorithmic solution.

An alternative approach [Bessani et al. 2013] to mitigate the risk of unavailability windows suggests that checkpoints can be coordinated across replicas so that at any given time there is only one replica in the checkpoint process. This approach takes advantage of the fault tolerance provided by the SMR architecture where a minority of faulty replicas do not compromise the system's operation. However, it does not address the localized unavailability window created at the replica running the checkpoint. Furthermore, the replica performing a checkpoint delays command execution until its completion. If the time spent to catch up with both delayed and newly submitted commands exceeds the checkpoint interval, multiple lagging replicas could still cause unavailability windows through delayed quorum.

Another approach [Zheng et al. 2014] suggests the creation of fuzzy checkpoints: a non-consistent snapshot of the system's state that can be created in parallel with command execution. A consistent state is then reconciled from one or more fuzzy checkpoints in addition to the command logs. This approach attacks the fundamental problem of creating unavailability windows by trading snapshot consistency for logging and recovery complexity.

3.3. Concurrent Tries (CTrie)

A Concurrent Trie (CTrie) [Prokopec et al. 2012] is a persistent non-blocking concurrent hash trie. Its interface, as originally presented, is an in-memory key-value data structure that provides consistent snapshots. In this discussion we concentrate on the fundamental primitives to write (insert and remove), read (lookup), and snapshot:

- $insert(k, v)$: given a key k, associates value v to it;
- $remove(k)$: given a key k, remove its associated value;
- $lookup(k)$: given a key k, returns its associated value;
- $snapshot()$: returns a logical copy of the data structure.

Functional persistent data structures allow for operations that return a logical copy of the data structure without performing an actual copy of all its elements, typically achieving logarithmic or even constant complexity [Okasaki 1999]. In a persistent trie, the data structure is updated by rewriting the path from its root to the leaf where the target key belongs to, leaving its remaining nodes intact. CTrie employs a similar approach with the addition of an indirect root node for non-blocking synchronization of concurrent modifications. The indirect node also contains a generation counter for snapshot purposes.

A consistent snapshot returns a state equivalent to the entire prefix set of operations up until the snapshot is taken – no intermediate states are observable. To ensure this property, CTrie implements a new procedure called "Generation Compare and Swap" (GCAS), based on the "Restricted Double-Compare Single-Swap" (RDCSS) introduced by [Harris et al. 2002]. GCAS takes 3 arguments: the indirect node observed at the beginning of the operation, the old node being replaced, and the new node to replace the old one. The GCAS semantics are similar to traditional "Compare and Swap" (CAS) operations except that it also compares the observed node's generation counter against the data structure's most recent generation.

When a snapshot is taken, it replaces its indirect root node with a new node of higher generation, returning the previous root node to the call-site. Ongoing write operations will fail due to the GCAS semantics and retry at the new generation while operations that finished before the snapshot will be accounted by it. Subsequent updates following a snapshot compare the generation counter form the target node's root with the data structure's most recent generation, performing necessary path rewrites at first access after a snapshot. Through the introduction of an indirect root node alongside with a generation counter and a customized compare-and-swap procedure, CTrie is able to share the necessary work to maintain snapshot consistency (i.e.: path rewriting) among updating threads while creating consistent snapshots in constant time ($\mathcal{O}(1)$).

4. Proposal

We propose the usage of a CTrie data structure to mitigate the checkpoint's impact during the normal operation of an SMR. Compared to [Zheng et al. 2014], using consistent checkpoints dispense reconciliation, thus being a fundamentally simpler model to adopt. This approach is orthogonal to the one presented by [Bessani et al. 2013] and could be used in addition to cross replica checkpoint coordination.

We regard CTrie as a generic structure to keep SMR's state. In order to be used for SMR checkpoints, additionally to being consistent, a CTrie snapshot has to assume a known position in the total order of execution provided by the atomic broadcast, therefore separating the prefix of commands before and after the snapshot so that logging can be consistent with the checkpoints taken.

4.1. Execution Model

We assume replicas are equipped with volatile memory and stable durable storage. Service state is kept in-memory. According to the SMR execution model of a replica (Algorithm 1), commands are delivered in total order at replicas with $deliver(i, m)$ (line 7). Each command is applied in its order, logged, and its response is sent to the client (lines 8 to 10). At each known interval Δ of commands, a checkpoint is performed by the replica and the log is reset to have only the commands after the last checkpoint (lines 12, 15, and 20).

Algorithm 1 presents both the traditional checkpoint (line 13) and this paper's proposal (line 16). In traditional checkpoint implementations, state has to be persisted to storage before executing new commands. In the proposed implementation, a CTrie snapshot serves as an SMR checkpoint. According to the CTrie definition, the snapshot operation marks the root of the CTrie with a new generation and subsequent state changes keep previous generation's state unchanged. Therefore, a snapshot operation provides a logical copy of the data structure in constant time, as discussed in Section 3.3. Commands following a snapshot take effect in the new generation of the CTrie, hence, keeping previous generation untouched and allowing the snapshot to be persisted concurrently with command execution on the new generation.

Considering the sequential SMR execution model and the CTrie semantics presented, commands executed before a snapshot are accounted by its generation while commands executed after a snapshot only take effect at the newly created generation. Hence, the CTrie snapshot has effect in a known position regarding the total order of commands

Algorithm 1 Replica's execution model.

```
1:  Types and Variables:
2:      state : the state of the replica (initially empty)        {in our approach: a CTrie}
3:      log : the log of commands applied (initially empty)
4:      Δ : the command interval to take checkpoints
5:      storage : represents the possibility of persisting data
6:  Replica works as follows:
7:      on deliver(i, cmd)                             {atomic broadcast delivers ith command}
8:          rsp ← apply(state, cmd)                      {sequentially applies cmd to state}
9:          append(log, cmd)                                              {add cmd to log}
10:         send(client(cmd), rsp)                      {assume client is known from cmd}
11:         if i mod Δ = 0 then                             {should start a checkpoint?}
12:             checkpoint(state, storage)                  {one of lines 13 or 16 below}
13: checkpointTraditional(state, storage)                              {traditional version}
14:     persist(state, storage)                             {writes state to stable storage}
15:     log ← {}                                                           {reset the log}
16: checkpointProposed(state, storage)                                    {proposed version}
17:     CTrie snpsht ← snapshot(state)    {a CTrie snapshot – it marks the root and returns}
18:     start thread to                                {concurrently with new commands}
19:         persist(snpsht, storage)                        {writes state to stable storage}
20:         log ← {}                                                       {reset the log}
```

which maintains the algorithm's linearization point – line 14 in the traditional SMR architecture and line 17 in the proposed solution. The adoption of the CTrie data structure preserves the sequential semantics of the SMR architecture, therefore adhering to its guarantees while mitigating the checkpoint's impact by moving data persistence to a separate thread and allowing for command execution to continue in parallel.

With regard to failure recovery, typically, a recovering replica: retrieves the most recent checkpoint and the associated log of commands, that is, commands that were already executed by the operational replicas but are not included in the retrieved checkpoint; installs the checkpoint; executes the obtained log of commands; and then resumes processing new commands arriving through the atomic broadcast primitive. Since the proposed approach generates logs consistent with checkpoints, this standard recovery procedure applies. Contributions in the literature to speed up the recovery process, such as [Bessani et al. 2013, Mendizabal et al. 2017], are also applicable although not discussed in this paper.

4.2. The SMaRtTrie Application

To fulfill the experimental evaluation proposed, we designed SMaRtTrie as an in-memory key-value storage. SMaRtTrie follows the SMR execution model presented in Section 4.1. Moreover, SMaRtTrie implements two state management models: the blocking (traditional) model, based on *TreeMap*[1]; and the concurrent (proposed) model, based on a *TrieMap*[2]. Both *TreeMap* and *TrieMap* provide in memory key-value stores supporting,

[1] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/TreeMap.html

[2] https://scala-lang.org/api/current/scala/collection/concurrent/TrieMap.html

besides key-value pair iterators, other commonly known operations for this kind of data structures:

- $get(k)$: Given a key k, returns its associated value, if any;
- $put(k, v)$: Given a key k, associates the value v to it;
- $remove(k)$: Given the key k, remove and return its associated value, if any.

Both models persist state by ensuring all its key-value pairs are transferred and written to durable storage. The prototype works with command batches, hence, the atomic broadcast primitive delivers the batch of commands processed at a given consensus instance. Commands in a batch are applied sequentially on the replicas state, following the SMR's execution model presented.

In order to focus on checkpoint's performance impact during normal operation, as well as to reduce performance variance in its evaluation, we assume failure free scenarios with disabled logging during the experiments. Although disabled, considering no changes to the SMR overall architecture were introduced, standard logging and failure recovery techniques still applies. Having both models implemented in SMaRtTrie, we present its experimental evaluation in Section 5.

4.2.1. Technologies and Platforms

The SMaRtTrie application has been developed using the Scala[3] programming language, which has a builtin CTrie data structure in its standard library called *TrieMap*. By leveraging Scala's interoperability with the Java[4] platform, SMaRtTrie is able to use the BFTS-MaRt [Bessani et al. 2014] library as the basis for its SMR implementation as well as the YCSB [Cooper et al. 2010] framework for benchmark purposes.

BFTSMaRt is a well-established library for the development of SMR applications in the Java platform. Like other SMR libraries, it offers standard operations to help with commonly required tasks such as checkpoints, command logging, state-transfer, and recovery.

YCSB is a benchmark tool that facilitates the comparison of different commonly used database service providers. YCSB comes with a set of standard benchmarks that can be easily integrated with most database services. Database vendors can implement the YCSB client interface and run its standard benchmarks to evaluate their products against different competitors. At the same time, infrastructure maintainers can run YCSB benchmarks to evaluate their infrastructure performance against the vendor's promises. Due to its high flexibility in customizing or creating entirely new benchmarks, the YCSB tool has been chosen to evaluate SMaRtTrie's performance in the experiments presented in this paper.

4.2.2. Common Engineering and Performance Aspects

To better evaluate the impact of the different approaches, we identified common sources of overhead and worked to minimize them so that performance results are more clearly

[3]`https://scala-lang.org`
[4]`https://www.java.com/`

associated to the different checkpoint approaches.

To minimize data copying during checkpoint execution, SMaRtTrie makes usage of memory mapped buffers[5] shared between its process and the operating system. SMaRtTrie's key-value pair encoder is capable of writing its binary format into memory mapped buffers, thus reducing needless allocation and data copying between application and the operating system during IO operations. SMaRtTrie also minimizes memory allocation of commonly used data structures in its binary encoder and decoder by employing object pools [Freeman 2015]. These optimizations equally benefit both the blocking and concurrent models as the optimized components play an equal role in both approaches.

The chosen runtime, the Java Virtual Machine (JVM) [Stärk et al. 2012], automatically manages memory allocation, retention, and release (a.k.a.: garbage collection or GC). At high level, a GC algorithm pauses the running program, frees its unused memory blocks, then resumes program execution. This process is commonly referred to as a GC event. Automatic memory management, although often associated with higher developer productivity, can impact memory intensive systems such as an in-memory data storage. Among the available choices[6] in the JVM's version 11, we emphasize here the algorithms evaluated for throughput impact during SMR operation. Detailed results are presented in Section 5.

- **G1 GC.** The G1 GC is the default algorithm in the JVM version 11. It's a general purpose GC that has a high probability of achieving GC goals while balancing high throughput;
- **Parallel GC.** The parallel collector is intended for medium to large-sized data sets that run on multi-core machines;
- **Z GC.** The Z garbage collector is indented for large memory sizes and applications that can not tolerate large GC pauses.

5. Evaluation

SMaRtTrie's performance has been evaluated experimentally in a proprietary cluster. The following subsections present the experimental environment, its parameters, and the results obtained.

5.1. Experimental Environment

The experimental environment used is a proprietary cluster composed by machines of various hardware profiles. Experiments presented in this paper were executed using two machines: a server running the SMaRtTrie application; and a client running a YCSB benchmark. Although SMR applications require multiple replicas to ensure fault tolerance, these experiments evaluate the performance of a single replica only. Considering that no changes to the SMR protocol were introduced, the results obtained are independent of cluster topology and should equally benefit additional replicas.

Both client and server have a Dual-Core AMD Opteron processor with 4 CPUs, 21 GB of memory, and 117 GB of SSD storage. Client and server also run the Ubuntu

[5] `https://man7.org/linux/man-pages/man2/mmap.2.html`

[6] `https://docs.oracle.com/en/java/javase/11/gctuning/available-collectors.html`

operating system version 18.04.1 and have the Java Virtual Machine version 11 installed. Machines in the cluster are connected via an internal high speed network. Despite the usage of a proprietary cluster, SMaRtTrie has no hardware specific requirements and should run with similar performance in environments of similar hardware profile.

A YCSB benchmark has been designed with three phases that run sequentially:

1. **Load:** inserts a number of random key-value pairs into the system;
2. **Warm-up:** runs a number of updates at random key-value pairs;
3. **Benchmark:** runs a number updates at random key-value pairs.

The number of key-value pairs as well as concurrent clients can be configured per phase. In the load phase, the benchmark creates a data set of configurable size to reduce the impact of expanding and/or re-balancing internal data structures during its execution. The warm-up phase is intended to heat caches and object pools to amortize their initialization impact into performance measurements. Finally, the benchmark phase executes a configured number of update requests for performance measurement. In both the warm-up and the benchmark phases, a configurable number of clients issue requests in a closed loop: a client issues a request and wait for its completion before submitting another. The YCSB framework reports performance measurements in the form of operations completed per second while SMaRtTrie reports checkpoint initialization and completion. The aggregated results are presented in the following sections.

5.2. Experiments' Parameters

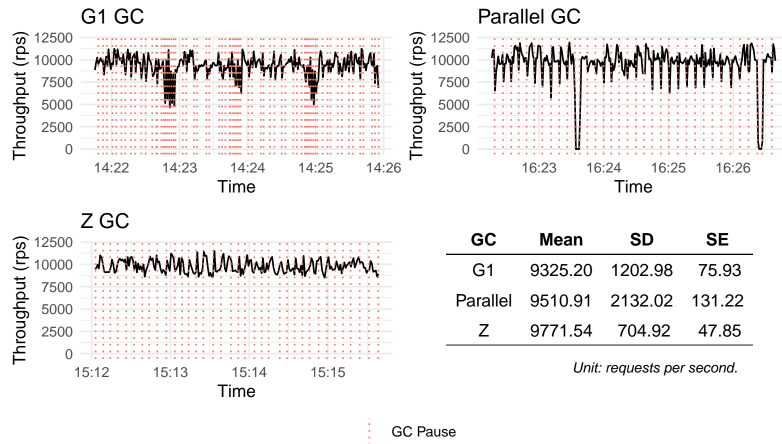
Preliminary experiments were executed for debugging and profiling purposes. The subsections bellow present the two major parameters choices derived from these experiments: the choice of garbage collection algorithm and the number of concurrent clients.

5.2.1. Garbage Collection Algorithm

The YCSB benchmark was executed with the SMaRtTrie prototype using different garbage collection algorithms in the Java Virtual Machine. These experiments were conducted without checkpoints. They reveal that throughput is severely impacted by the choice of the garbage collection algorithm. Figure 1 shows a comparison of throughput by GC, including its mean, standard deviation (SD), and error (SE) measurements.

We observe a correlation between throughput drops and GC pauses while using the G1 garbage collector, particularly related to periods of high GC activity. Although GC pauses are sparse when using the Parallel GC algorithm, it occasionally produces larger pauses which translates into significant drops in the system's throughput. The Z GC algorithm issues regular but short pauses which managed to produce the least impact in our experiments, as evidenced by its higher average throughput, lower standard deviation, and lower standard error.

Z GC was chosen for the remaining experiments presented in this paper. Its low throughput impact further increases confidence in the comparisons between the blocking and concurrent models by reducing environmental noise observed during experimental evaluation.



GC	Mean	SD	SE
G1	9325.20	1202.98	75.93
Parallel	9510.91	2132.02	131.22
Z	9771.54	704.92	47.85

Unit: requests per second.

Figure 1. Throughput by garbage collection algorithm.

5.2.2. Number of Concurrent Clients

YCSB clients generate a workload by continuously issuing sequential requests in a closed loop routine. Each client runs in its own thread. An experiment with increasing number of concurrent clients was executed in order to estimate the system's saturation point (Fig. 2). The system's peak throughput is achieved with 32 clients. Beyond this threshold, throughput does not increase with the number of concurrent clients but latency increases significantly. We conservatively chose to run the remaining experiments with 8 clients so that they stay safely bellow its saturation mark.

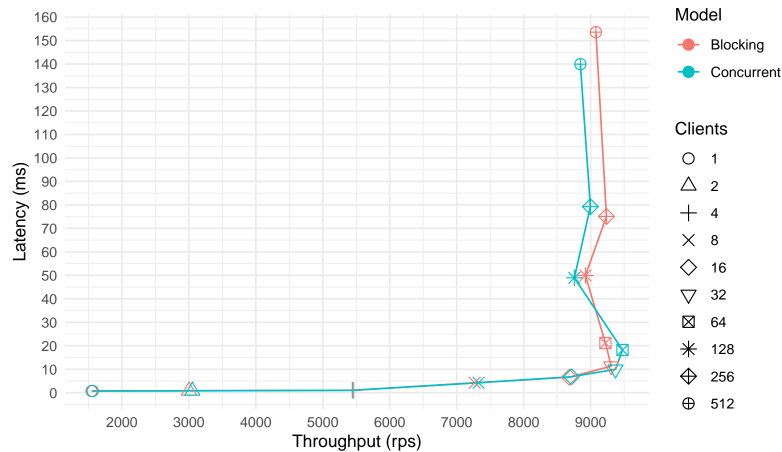


Figure 2. Throughput vs latency by number of clients.

Additionally, a micro-benchmark was executed with the data structures used in the prototype to determine their relationship with the system's saturation point. The micro-benchmark is a single threaded process executing a series of updates in each target data structure after an initial data load. Figure 3 shows that the throughput of the data structures in isolation are far greater than then system's peak throughput, indicating that state management is not a limiting factor at current scale.

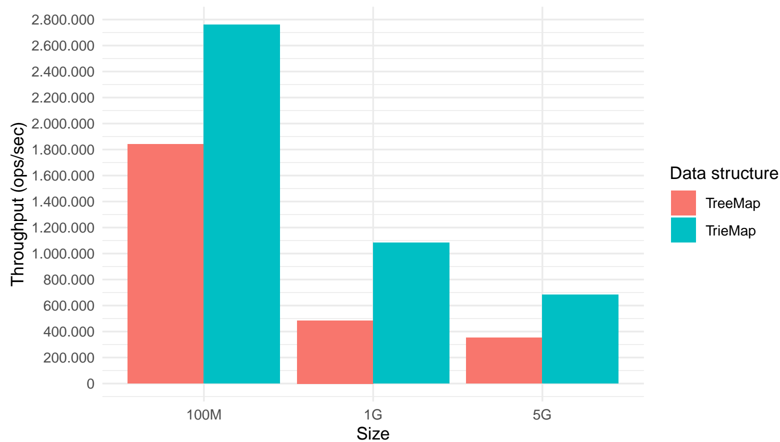


Figure 3. Data structures micro-benchmarks. Throughput vs. state size.

5.3. Results

Figure 4 shows the throughput of the blocking checkpoint model with data set of 5 GB while running 20M requests. As described in Section 3.2, to keep consistency, the application stops processing commands during checkpoint execution which are delimited by the red (start) and green (stop) markers. This leads to the impacts measured, showing sudden drops in the system's throughput that last for the duration of the checkpoint process. This behavior has been observed in the literature [Bessani et al. 2013] and is hereby reproduced.

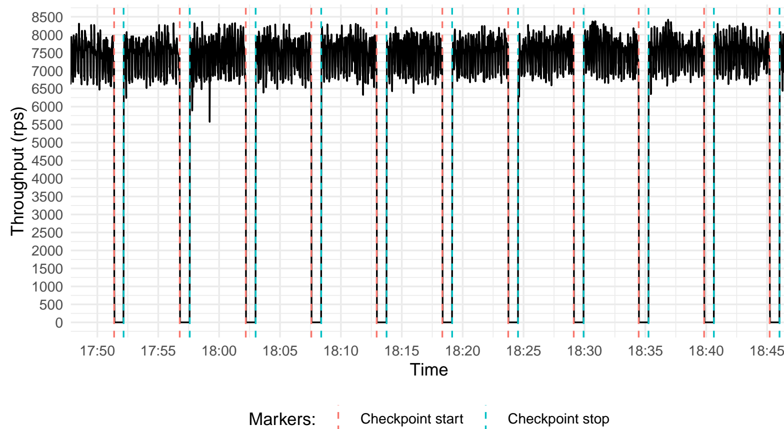


Figure 4. Throughput using blocking checkpoints.

Figure 5 shows the throughput of the concurrent state management model with same data set size and number of requests (5G and 20M). As described in Sections 3.3 and 4.1, the CTrie's capabilities that allows for the checkpoint process to run in parallel with command execution alleviates the checkpoint's impact into the system's requests per second metric. As before, the red and green markers show respectively the moment the checkpoint is requested and finishes being persisted into durable storage.

The request rate sustained by the concurrent model during checkpoint periods produces overall higher average throughput, lower standard deviation (SD), and lower

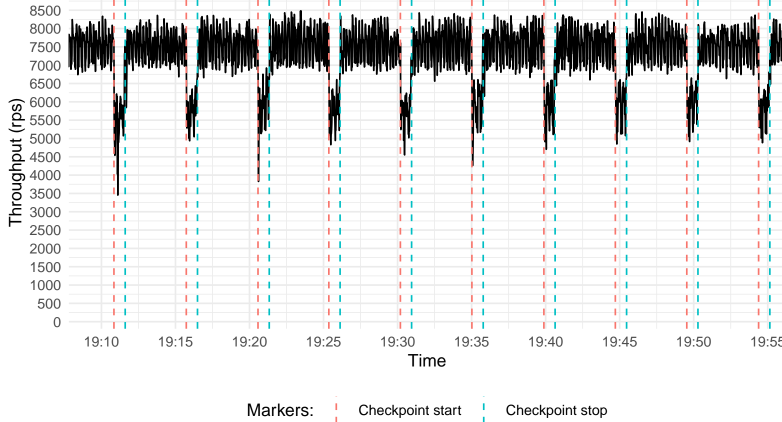


Figure 5. Throughput using concurrent checkpoints.

standard error (SE) (Tab. 1). Hence, the concurrent model is not only faster on average than its blocking counterpart but, it also responds to client requests with less latency variability overtime. A further breakdown of throughput observed outside (No) and during (Yes) checkpoint periods is shown in Table 2. It reveals that the concurrent model is able to sustain about 78% of its normal throughput during checkpoints. There isn't an equivalent breakdown for the blocking model as its throughput during checkpoints its trivially observed to be zero (Fig. 4).

Model	Mean	SD	SE
Blocking	6375.02	2685.29	44.75
Concurrent	7269.79	780.73	13.01
±	+14.04%	-70.93%	-70.93%

Table 1. Overall throughput (request/second) by model.

Checkpoint	Mean	SD	SE
No	7518.87	494.47	8.96
Yes	5894.44	625.36	26.62
±	-21.6%	-26.47%	+197.18%

Table 2. Concurrent model's throughput (request/second) by period.

Finally, Figure 6 presents average latency while varying the number of clients and state size. Both models increase their throughput with the number of concurrent clients. The smallest state size (100M) present similar average throughput in both models. However, as the state size grows, and therefore the checkpoint process becomes more expensive and time-consuming, the concurrent checkpoint model starts to present higher average throughput.

The experiments presented in this section show that SMaRtTrie's concurrent checkpoint model allows it to outperform its traditional blocking checkpoint counterpart. Although evaluations were made at a single replica, its benefits should equally extend

Figure 6. Throughput by thread and state size.

to additional replicas. Moreover, the concurrent model addresses the fundamental problem of stopping command processing during checkpoint execution which, as described in Section 3.2, could cause an unavailability window if a majority of replicas start their checkpoint process in an overlapping time period. The work to measure the performance benefits of deploying SMaRtTrie in a multi-replica topology is left for future contributions.

6. Conclusion

This paper presents SMaRtTrie: a simple in-memory key-value store based on a CTrie data structure. We demonstrated the benefit of its concurrent checkpoint model via experimental evaluation. Noticeably, SMaRtTrie is able to reach the same throughput as other commonly used solution while sustaining 78% throughput during checkpoints. We have discussed how consistent concurrent checkpoints benefits multi-replica SMR applications, although its evaluation is left for future contributions. Additionally, an orthogonal research into the impact of garbage collection algorithms was presented while defining the experiments' parameters.

The data structures used in the prototype have shown high throughput in isolation (Sec. 5.2). However, the system peak throughput is nowhere near the data structures' peak performance, suggesting that there are other limiting components. In this paper we focused on the impact of checkpoints in SMR systems, leaving additional optimizations to non-checkpoint related components to future contributions. SMaRtTrie uses a general purpose CTrie implemented in the Scala language's standard library. It's feasible to conceive that a specialized version of a CTrie built with the purpose of supporting in-memory key-value SMR systems could expand its benefits further.

SMaRtTrie's source code along with its experiment instrumentation, results, and analysis are available via the GitHub [7] open source platform.

References

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*

[7] `http://github.com/erickpintor/smarttrie`

({USENIX}{ATC} 13)*, pages 169–180.

Bessani, A., Sousa, J., and Alchieri, E. E. (2014). State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE.

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Freeman, A. (2015). The object pool pattern. In *Pro Design Patterns in Swift*, pages 137–156. Springer.

Harris, T. L., Fraser, K., and Pratt, I. A. (2002). A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*, pages 265–279. Springer.

Herlihy, M. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

Lamport, L. et al. (2001). Paxos made simple. *ACM Sigact News*, 32(4):18–25.

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.

Okasaki, C. (1999). *Purely functional data structures*. Cambridge University Press.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319.

Prokopec, A., Bronson, N. G., Bagwell, P., and Odersky, M. (2012). Concurrent tries with efficient non-blocking snapshots. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 151–160.

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

Stärk, R. F., Schmid, J., and Börger, E. (2012). *Java and the Java virtual machine: definition, verification, validation*. Springer Science & Business Media.

Zheng, W., Tu, S., Kohler, E., and Liskov, B. (2014). Fast databases with fast durability and recovery through multicore parallelism. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 465–477.