High Availability For Key-Value Stores Using Checkpoint/Restore

Fadhil Abubaker, Hussain Sadiq Abuwala

Abstract

High availability (HA) in distributed systems is achieved through replication. However, correctly implementing HA within a system is often a difficult task, resulting in increased code complexity and performance degradation. Existing work has looked at how a system can offload replication to an external subsystem, such as the virtual machine (VM) layer. In this paper, we present a novel external replication technique that uses process-based checkpoint/restore to guarantee HA. We build a simple key-value store on top of this replication layer and benchmark it against an implementation that uses asynchronous statement replication. Results show that this technique does not incur much overhead and can be a viable alternative to implementing replication within the database system.

1 Introduction

High availability (HA) is an important requirement for modern distributed systems. HA is achieved through replication, where multiple replicas of the system are run on different nodes for redundancy. Updates made to one replica are propagated to the others in a synchronous or asynchronous manner. Additionally, replication can be active-active, where any replica can accept updates or active-passive, where only a master replica can accept updates [5]. Thus, the failure of one replica does not affect the operation of the entire system, as another replica can take its place, and continue serving requests.

However, implementing HA within a distributed system is a challenging task. As mentioned in [8], to build active-standby replication into a database management system (DBMS), the system has to implement propagating updates from the active replica to the standby, coordinate transactions between the replicas and ensure atomic handover from active to standby in the face of a failure. Moreover, these components have to be carefully implemented so as to have minimal impact on the performance of the underlying system.

Given the complexity of implementing HA, the question arises whether it should be pushed outside of the DBMS. Existing work has looked at whether delegating replication to an external subsystem is acceptable in exchange for a simplified implementation. One such subsystem is the storage layer itself, where a shared-disk architecture can be used to host the database. It was even shown that a highly-available, shared-disk setup can offer greater performance than a standalone setup that does not provide high availability [6]. Another subsystem is the virtual machine (VM) layer, which can be configured to replicate changes between a primary and secondary VM [2, 4, 13]. Prior work has looked at how to adapt DBMSs to use VM replication for HA without compromising on performance [8].

In this paper, we design and build a key-value store that uses process-based checkpoint/restore for active-standby replication. Our approach is similar to VM replication, except the granularity of replication is at the process level. In a nutshell, we capture regular snapshots of the process state of the active replica, migrate it to the standby, and restore from the snapshot as part of failover. We implement this using Checkpoint/Restore In Userspace (CRIU) [3], a Linux-based utility that can be used to checkpoint and restore the runtime of a process.

The rest of the paper is organized as follows: we first outline some background on replication techniques used in modern key-value stores, using Redis [9] as a driving example, then provide an overview of our system and finally, evaluate its performance using a standardized benchmark.

2 Background

2.1 Redis Replication

In-memory key-value stores like Redis are used for low-latency, high-throughput tasks that are ephemeral in nature, such as web caching, session management, etc. Replicating data in these use-cases is not of the utmost importance, but will help prevent spikes in performance after failover. Redis implements HA through statement-based replication, where commands from the client are executed on the active replica and streamed to the standby. It has built-in active-standby replication and by default, performs this replication asynchronously.

However, there is a caveat: Redis replication requires persistence to be activated on the master, which in turn can lower the performance of the in-memory model [12]. This is because if persistence is turned off, and the master crashes and recovers quickly, it will restart with an empty dataset. This will then be streamed to the standby replica, which will effectively truncate the data on the standby.

Thus Redis replication necessitates persistence. In the next section, we describe the persistence modes Redis has, which will tie-in to the replication technique used in our system.

2.2 Redis Persistence

Redis supports two types of persistence [10]:

- RDB (Redis Database)
- AOF (Append-Only File)

RDB involves performing point-in-time snapshots of a particular dataset at specified regular intervals. When the user-specified checkpoint interval is reached, Redis forks a child process, which writes the dataset to a temporary RDB file. Once the new RDB file is completely written, the old one is replaced. Note that RDB is prone to some data loss since any updates made after a snapshot will be lost if the server crashes before the next snapshot.

AOF, on the other hand, circumvents this problem by writing every operation received by the server into the AOF persistence log. However, this approach significantly degrades performance as every write will have to be flushed to disk, blunting the advantages of an in-memory store like Redis.

Our replication technique is similar to RDB, in that we capture regular snapshots of the key-value store and save it to disk. However, unlike Redis, which uses statement-based replication and can use RDB snapshots for persistence, our system uses snapshots for both replication and persistence.

3 System Architecture

3.1 Overview

Our system architecture is briefly sketched in Fig 1. It consists of three nodes, where two of them must be connected to a shared-disk; these are the nodes that will host the active and standby replicas. The active and standby nodes each contain three processes: the **server**, the **replicator** and the **checkpoint/restore daemon**. The third node hosts the **rproxy**, which is a simple reverse proxy that routes requests to the server on the currently active node. We now expand on each component below.

3.1.1 Server

The server process is the actual key-value store that can accept and serve requests. The implementation of the server is kept simple so we can focus on building the replication technique. The server exposes two APIs for requests: `Put(key, value)` and `Get(key)`. Note that `value` here is not an atomic type, rather, it is a set of fields that each hold a value. In our implementation, we treat `value` as a JSON object. Clients communicate with the server over HTTP. For example, retrieving a key would involve sending a GET request to `http://server-ip/key`. Inserting a key is similar, except a PUT request is used and the values are encoded in the request body. Internally, we parse the request body and store the keys and their values in an in-memory hash table. The server also keeps track of

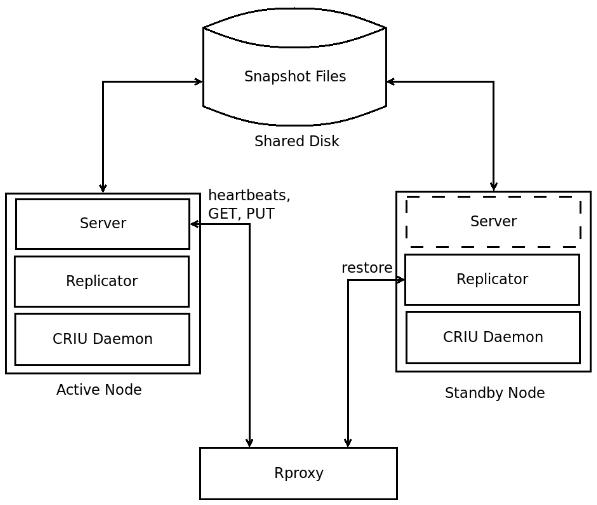

Figure 1: System Architecture.

the number of PUT requests it has received, and once a certain threshold is reached, it will notify the replicator to capture a snapshot of itself. This is similar to how Redis RDB works, where the user can specify a required number of writes made to the dataset after which an RDB file will be created.

3.1.2 Replicator

The replicator is a thin process that acts as a wrapper around the checkpoint/restore daemon. It exposes two APIs to control replication, `checkpoint` and `restore`. The former captures a snapshot of the server and stores it on the shared-disk, while the latter retrieves the latest snapshot from the shared-disk and restores it. It does this by sending RPC calls to the checkpoint/restore daemon. The `checkpoint` method is called by the server when the checkpoint interval has passed, and the `restore` method is called by the rproxy, when it detects that the server on the active node is down.

3.1.3 Checkpoint/Restore Daemon

The checkpoint/restore daemon is a daemonized version of the CRIU tool. External processes (such as the replicator) communicate with it through RPC calls over a Unix socket. The daemon is responsible for the system-level code that checkpoints and restores the process state of the server. A more elaborate overview of CRIU is given in section 3.2.

3.1.4 Rproxy

The rproxy is the client-facing process that is responsible for routing requests to the active replica. On initialization, it connects to the server on the active node and sends regular heartbeats to it. If it detects from a lapsed heartbeat check that the server is down, it calls `restore` on the replicator of the standby node to restore the server from the latest snapshot. Subsequently, the former standby now becomes the active replica.

3.2 Checkpoint/Restore In Userspace

The CRIU utility can be used to freeze the state of a running process/container and save this state to a set of image files on persistent storage. The process can then be restored back to the point it was frozen from the image files. It is integrated in multiple container runtimes such as OpenVZ, LXC/LXD, Docker, and Podman. For example, Google's Borg cluster management system uses CRIU for live-migrating jobs across their clusters [7].

CRIU serializes the contents of a process's virtual memory address into image files. It does so by obtaining the memory pages currently in use by the process using the `/proc` filesystem on Linux. Once it has identified the pages, it injects parasite code into the process using `ptrace` and drains the contents of its memory into files on disk. On restore, CRIU forks itself, applies the contents of the images files into its memory address and resumes execution. The image files can be copied to a different machine and then restored, making CRIU ideal for moving or replicating processes across machines.

4 Evaluation

4.1 Methodology

We use the Yahoo! Cloud Serving Benchmark (YCSB), a popular key-value store benchmark suite for evaluating our system. It automates essential benchmarking tasks such as workload definition, workload execution and performance measurement. To add our key-value store to YCSB's suite, we implemented a custom database interface layer that uses our key-value store's API. In particular, we implemented the `read`, `update`, `insert` and `delete` methods, omitting `scan` due to time constraints. We use workloads A and B from the YCSB suite, each of which execute 1000 operations and have a read/write mix of 50/50 and 95/5 respectively. For replication, we configured the server to capture snapshots after every 250 updates.

Our experiments were run on a cluster of three nodes, each with 2 Intel Xeon E5-2620v2 processors for a total of 24 cores, and 32 GB of RAM. The active and standby nodes are also connected to an NFS drive with 4TB of storage, which is used as the shared-disk for storing snapshots.

4.2 Storage Performance vs Redis

To assess the baseline performance of our key-value store's storage engine, we disabled replication and ran YCSB workloads A and B and compared it with Redis. The results are shown in figure 2.

Note that since our focus was on the replication layer and not the storage engine itself, we expected the performance to be subpar compared to Redis. We believe the overhead primarily comes from using HTTP as the message protocol and the serialization/de-serialization of JSON objects in each request. Comparatively, Redis uses the Redis Serialization Protocol (RESP) [11] and an encoding scheme that is fast to parse which possibly results in an order of magnitude better performance.

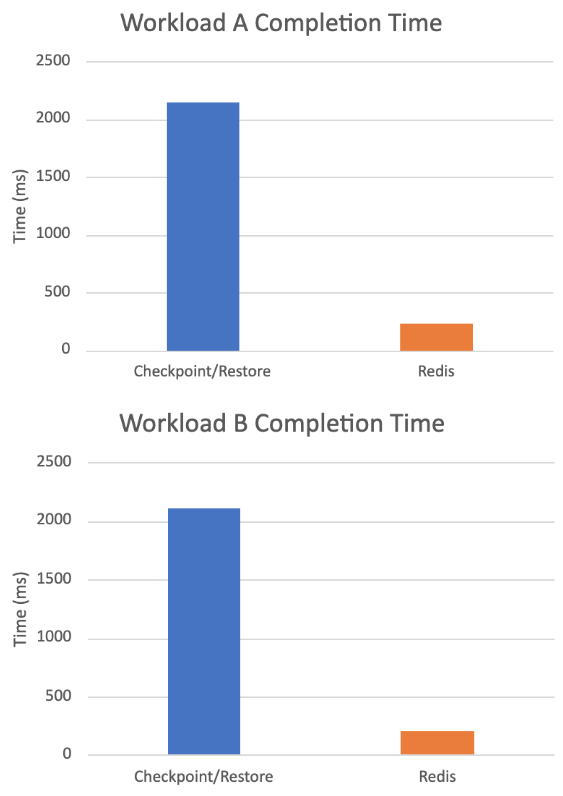

Figure 2: Workload completion times for workloads A and B against Redis.

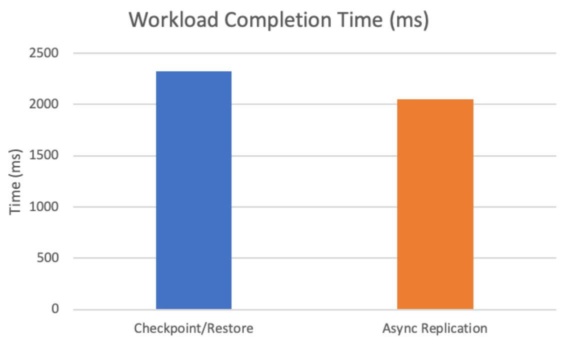

Figure 3: Workload completion times for workload A against asynchronous replication.

4.3 Replication Performance vs Asynchronous Replication

We compare the performance of snapshot-based replication against asynchronous statement-based replication. We do this by implementing the latter in our key-value store by shipping the statement to the standby replica, after the server has replied to the client. Note that we only use workload A for the comparison, since workload B consists of 95% read requests which are not enough to trigger snapshots, causing the performance of checkpoint/restore to be the same as asynchronous replication.

Figure 3 shows the workload completion times by each replication technique. Given that checkpoint/restore captures snapshots after every 250 updates, we did expect a slight increase in completion time compared to asynchronous replication. Recall that workload A has a 50/50 read/write mix, which results in around 500 update operations, giving us 2 snapshots for this run.

Figure 4 shows the average and max latency for read and update requests by each replication technique. Average latencies for both types of requests are around the same; this can be explained by the fact that the cost of snapshotting is amortized across multiple requests. While max latencies for update requests are the same, max latencies for read requests have a much larger discrepancy: a wide gap exists due to unlucky read requests being blocked as a snapshot is in progress.

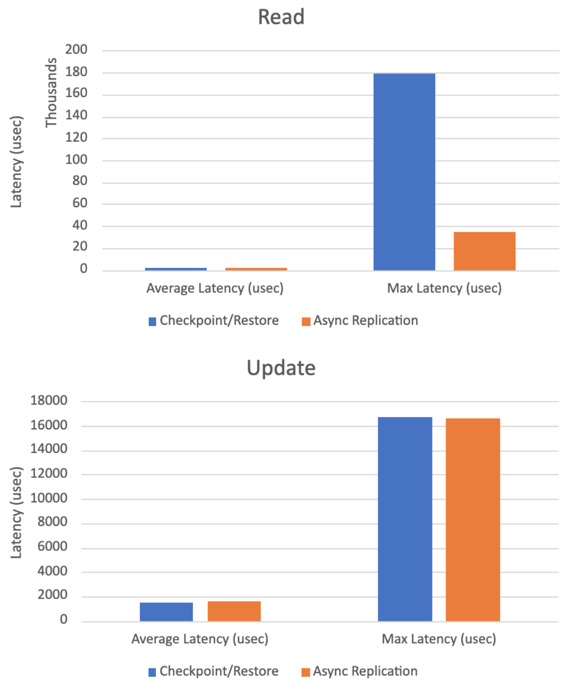

Figure 4: Average and max latency for read and update requests against asynchronous replication.

4.4 Impact of Checkpoint Intervals on Workload Completion

Checkpointing the state of the key-value store is an expensive operation. More frequent checkpoints will minimize data loss due to a crash but at the cost of performance. To assess the impact of the checkpoint interval on the performance of the key-value store, we vary the number of writes that trigger a checkpoint and run a custom workload consisting of 5000 insert operations. The results are shown in figure 5. As expected, workload completion times decrease as checkpoint intervals increase. This comes with an increase in possible data loss, hence the exact interval will be

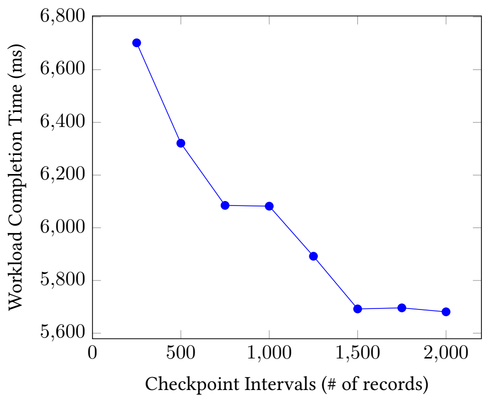

Figure 5: Checkpoint intervals vs workload completion time for a workload with 5000 insert operations.

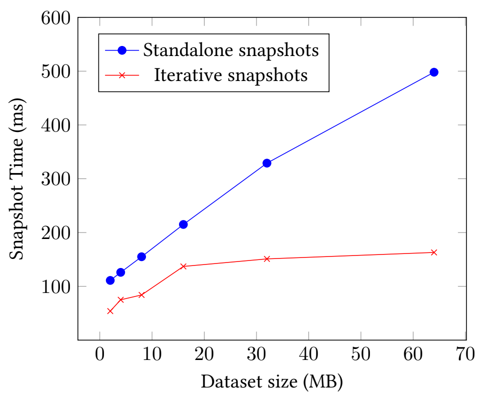

Figure 6: Time taken for standalone vs iterative snapshots across increasing dataset sizes.

application-dependent and can be tuned by the user.

4.5 Standalone vs Iterative Snapshots

CRIU snapshots can be standalone, where the entire memory address space of the process is checkpointed as is, or iterative, where snapshots only capture memory pages that have changed since the last snapshot. The differences in time taken to complete a snapshot as the dataset size increases can be seen in figure 6. Using standalone snapshots for frequent checkpoints would be inefficient, since the snapshot time would be proportional to the size of the process's memory. Iterative snapshots are more efficient, since they capture a constant amount of data between snapshots. Our implementation uses iterative snapshots to minimize the time that requests are blocked while a snapshot is in progress.

5 Conclusion

In this paper, we have explored using checkpoint/restore for replication to guarantee HA. To that extent, we implemented a key-value store that uses checkpoint/restore to replicate its in-memory state to another machine. Initial results show that this approach works well with minimal overhead compared to asynchronous statement-based replication. Further optimizations and improvements can be made to checkpoint/restore which can result in better performance and durability.

For example, it is possible to move from a shared-disk to a shared-nothing setup by utilizing CRIU's page server feature. This allows the active replica to stream image files across the network to a page server on the standby replica.

The current implementation of checkpoint/restore guarantees RDB-style durability, which can lose data if the server crashes in between snapshots. For stronger durability, we can implement *output commit* as mentioned in [8], where output packets are buffered until the next snapshot. This would turn checkpoint/restore into a more synchronous replication technique, where greater durability comes at the price of higher request latency.

Currently, the standby replica does not actually run a server process until failover. CRIU has no existing support for applying images to a running process, but it is a planned feature [1]. As a result, it is not viable to convert the standby to serve read requests: new snapshots would have to kill the running server on the standby and then restore it from the latest snapshot, leading to frequent client disconnections. Once the apply images feature is introduced, checkpoint/restore can be extended to have read requests be served by the standby.

Thus, checkpoint/restore is a potentially viable replication technique with acceptable tradeoffs between replication freedom and implementation complexity. By delegating replication to an external layer, database systems can achieve high availability with minimal implementation and performance overhead.

References

[1] Applying images - CRIU. URL: `https://criu.org/Applying_images` (visited on 03/06/2022).

[2] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. *SIGOPS Oper. Syst. Rev.*, 29(5):1–11, Dec. 1995. ISSN: 0163-5980.

[3] CRIU website. URL: https://criu.org/Main_Page (visited on 03/06/2022).

[4] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 161–174, San Francisco, California. USENIX Association, 2008. ISBN: 1119995555221.

[5] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 173–182, Montreal, Quebec, Canada. Association for Computing Machinery, 1996. ISBN: 0897917944.

[6] J. Kim, K. Salem, K. Daudjee, A. Aboulnaga, and X. Pan. Database high availability using shadow systems. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 209–221, Kohala Coast, Hawaii. Association for Computing Machinery, 2015. ISBN: 9781450336512.

[7] V. Marmol and A. Tucker. Task migration at scale using CRIU. In Linux Plumbers Conference, 2018.

[8] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield. Remusdb: transparent high availability for database systems. *Proc. VLDB Endow.*, 4(11):738–748, Aug. 2011. ISSN: 2150-8097.

[9] Redis. URL: https://redis.io/ (visited on 04/01/2022).

[10] Redis persistence. URL: https://redis.io/docs/manual/persistence/ (visited on 04/01/2022).

[11] Redis protocol. URL: https://redis.io/docs/reference/protocol-spec/ (visited on 04/01/2022).

[12] Redis replication. URL: https://redis.io/docs/manual/replication/ (visited on 04/01/2022).

[13] D. J. Scales and G. Venkitachalam. The design and evaluation of a practical system for fault-tolerant virtual machines. *Operating Systems Review*, 2010.

6 Appendix

6.1 Post-mortem

- The project was quite implementation heavy, since we had to implement the key-value store API, figure out the internals of CRIU and figure out how to run YCSB to get benchmark results.

- The CRIU daemon API uses gRPC and its examples were documented in Go, because of this, we implemented our system using Go. Since one of the authors has experience with Go, this was not too much of an impediment.

- CRIU has interesting research applications that we were eager to explore, but could not because of time constraints. For example, it supports migrating existing TCP connections from one machine to the other. This has some potential use-cases for live-migration scenarios.

- The benchmarks were run on the DSG group's tembo cluster. The /home directories on tembo are shared NFS drives, which made deploying our system easier.

- We initially wanted to run checkpoint/restore on top of an existing key-value store like Redis, but there were some issues with CRIU that made it difficult to checkpoint Redis, so due to time constraints we implemented a simple key-value store to drive our replication technique.

- We initially wanted to replicate at the container-level as opposed to the process-level, but existing container runtimes do not fully support some CRIU features like iterative snapshotting, so we decided to use CRIU at the process-level.