On the Efficiency of Durable State Machine Replication

Alysson Bessani[1], *Marcel Santos*[1], *João Felix*[1], *Nuno Neves*[1], *Miguel Correia*[2]
{[1]FCUL/LaSIGE, [2]INESC-ID/IST}, University of Lisbon – Portugal

Abstract

State Machine Replication (SMR) is a fundamental technique for ensuring the dependability of critical services in modern internet-scale infrastructures. SMR alone does not protect from full crashes, and thus in practice it is employed together with secondary storage to ensure the durability of the data managed by these services. In this work we show that the classical durability enforcing mechanisms – logging, checkpointing, state transfer – can have a high impact on the performance of SMR-based services even if SSDs are used instead of disks. To alleviate this impact, we propose three techniques that can be used in a transparent manner, i.e., without modifying the SMR programming model or requiring extra resources: parallel logging, sequential checkpointing, and collaborative state transfer. We show the benefits of these techniques experimentally by implementing them in an open-source replication library, and evaluating them in the context of a consistent key-value store and a coordination service.

1 Introduction

Internet-scale infrastructures rely on services that are replicated in a group of servers to guarantee availability and integrity despite the occurrence of faults. One of the key techniques for implementing replication is the *Paxos* protocol [27], or more generically the *state machine replication* (SMR) approach [34]. Many systems in production use variations of this approach to tolerate *crash faults* (e.g., [4, 5, 8, 12, 19]). Research systems have also shown that SMR can be employed with *Byzantine faults* with reasonable costs (e.g., [6, 9, 17, 21, 25]).

This paper addresses the problem of adding durability to SMR systems. *Durability* is defined as the capability of a SMR system to survive the crash or shutdown of all its replicas, without losing any operation acknowledged to the clients. Its relevance is justified not only by the need to support maintenance operations, but also by the many examples of significant failures that occur in data centers, causing thousands of servers to crash simultaneously [13, 15, 30, 33].

However, the integration of durability techniques – logging, checkpointing, and state transfer – with the SMR approach can be difficult [8]. First of all, these techniques can drastically decrease the performance of a service[1]. In particular, *synchronous logging* can make the system throughput as low as the number of appends that can be performed on the disk per second, typically just a few hundreds [24]. Although the use of SSDs can alleviate the problem, it cannot solve it completely (see §2.2). Additionally, *checkpointing* requires stopping the service during this operation [6], unless non-trivial optimizations are used at the application layer, such as copy-on-write [8, 9]. Moreover, recovering faulty replicas involves running a *state transfer protocol*, which can impact normal execution as correct replicas need to transmit their state.

Second, these durability techniques can complicate the programming model. In theory, SMR requires only that the service exposes an *execute()* method, called by the replication library when an operation is ready to be executed. However this leads to logs that grow forever, so in practice the interface has to support service state checkpointing. Two simple methods can be added to the interface, one to collect a snapshot of the state and another to install it during recovery. This basic setup defines a simple interface, which eases the programming of the service, and allows a complete separation between the replication management logic and the service implementation. However, this interface can become much more complex, if certain optimizations are used (see §2.2).

This paper presents new techniques for implementing data durability in crash and Byzantine fault-tolerant

[1]The performance results presented in the literature often exclude the impact of durability, as the authors intend to evaluate other aspects of the solutions, such as the behavior of the agreement protocol. Therefore, high throughput numbers can be observed (in req/sec) since the overheads of logging/checkpointing are not considered.

(BFT) SMR services. These techniques are transparent with respect to both the service being replicated and the replication protocol, so they do not impact the programming model; they greatly improve the performance in comparison to standard techniques; they can be used in commodity servers with ordinary hardware configurations (no need for extra hardware, such as disks, special memories or replicas); and, they can be implemented in a modular way, as a *durability layer* placed in between the SMR library and the service.

The techniques are three: *parallel logging*, for diluting the latency of synchronous logging; *sequential checkpointing*, to avoid stopping the replicated system during checkpoints; and *collaborative state transfer*, for reducing the effect of replica recoveries on the system performance. This is the first time that the durability of fault-tolerant SMR is tackled in a *principled* way with a set of algorithms organized in an *abstraction* to be used between SMR protocols and the application.

The proposed techniques were implemented in a durability layer on the BFT-SMaRt state machine replication library [1], on top of which we built two services: a consistent key-value store (SCKV-Store) and a non-trivial BFT coordination service (Durable DepSpace). Our experimental evaluation shows that the proposed techniques can remove most of the performance degradation due to the addition of durability.

This paper makes the following contributions:

1. A description of the performance problems affecting durable state machine replication, often overlooked in previous works (§2);

2. Three new algorithmic techniques for removing the negative effects of logging, checkpointing and faulty replica recovery from SMR, without requiring more resources, specialized hardware, or changing the service code (§3).

3. An analysis showing that exchanging disks by SSDs neither solves the identified problems nor improves our techniques beyond what is achieved with disks (§2 and §5);

4. The description of an implementation of our techniques (§4), and an experimental evaluation under write-intensive loads, highlighting the performance limitations of previous solutions and how our techniques mitigate them (§5).

2 Durable SMR Performance Limitations

This section presents a durable SMR model, and then analyzes the effect of durability mechanisms on the performance of the system.

2.1 System Model and Properties

We follow the standard SMR model [34]. Clients send requests to invoke operations on a service, which is implemented in a set of replicas (see Figure 1). Operations are executed in the same order by all replicas, by running some form of agreement protocol. Service operations are assumed to be deterministic, so an operation that updates the state (abstracted as a *write*) produces the same new state in all replicas. The state required for processing the operations is kept in main memory, just like in most practical applications for SMR [4, 8, 19].

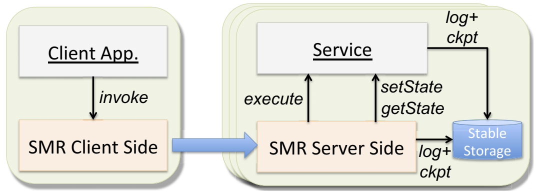


Figure 1: A durable state machine replication architecture.

The replication library implementing SMR has a client and a server side (layers at the bottom of the figure), which interact respectively with the client application and the service code. The library ensures standard safety and liveness properties [6, 27], such as correct clients eventually receive a response to their requests if enough synchrony exists in the system.

SMR is built under the assumption that at most f replicas fail out of a total of n replicas (we assume $n = 2f + 1$ on a crash fault-tolerant system and $n = 3f + 1$ on a BFT system). A crash of more than f replicas breaks this assumption, causing the system to stop processing requests as the necessary agreement quorums are no longer available. Furthermore, depending on which replicas were affected and on the number of crashes, some state changes may be lost. This behavior is undesirable, as clients may have already been informed about the changes in a response (i.e., the request completed) and there is the expectation that the execution of operations is persistent.

To address this limitation, the SMR system should also ensure the following property:

> *Durability:* Any request completed at a client is reflected in the service state after a recovery.

Traditional mechanisms for enforcing durability in SMR-based main memory databases are logging, checkpointing and state transfer [8, 16]. A replica can recover from a crash by using the information saved in stable storage and the state available in other replicas. It is important to notice that a recovering replica is considered faulty *until it obtains enough data to reconstruct the state* (which typically occurs after state transfer finishes).

Logging writes to stable storage information about the progress of the agreement protocol (e.g., when certain messages arrive in Paxos-like protocols [8, 20]) and about the operations executed on the service. Therefore, data is logged either by the replication library or the service itself, and a record describing the operation has to be stored before a reply is returned to the client.

The replication library and the service code synchronize the creation of checkpoints with the truncation of logs. The service is responsible for generating snapshots of its state (method *getState*) and for setting the state to a snapshot provided by the replication library (method *setState*). The replication library also implements a *state transfer* protocol to initiate replicas from an updated state (e.g., when recovering from a failure or if they are too late processing requests), akin to previous SMR works [6, 7, 8, 9, 32]. The state is fetched from the other replicas that are currently running.

2.2 Identifying Performance Problems

This section discusses performance problems caused by the use of logging, checkpointing and state transfer in SMR systems. We illustrate the problems with a consistent key-value store (SCKV-Store) implemented using BFT-SMaRt [1], a Java BFT SMR library. In any case, the results in the paper are mostly orthogonal to the fault model. We consider write-only workloads of 8-byte keys and 4kB values, in a key space of 250K keys, which creates a service state size of 1GB in 4 replicas. More details about this application and the experiments can be found in §4 and §5, respectively.

High latency of logging. As mentioned in §2.1, events related to the agreement protocol and operations that change the state of the service need to be logged in stable storage. Table 1 illustrates the effects of several logging approaches on the SCKV-Store, with a client load that keeps a high sustainable throughput:

Metric	No log	Async.	Sync. SSD	Sync. Disk
Min Lat. (ms)	1.98	2.16	2.89	19.61
Peak Thr. (ops/s)	4772	4312	1017	63

Table 1: Effect of logging on the SCKV-Store. Single-client minimum latency and peak throughput of 4kB-writes.

The table shows that *synchronous*[2] logging to disk can cripple the performance of such system. To address this issue, some works have suggested the use of faster non-volatile memory, such as flash memory solid state drives (SSDs) or/in NVCaches [32]. As the table demonstrates, there is a huge performance improvement when the log is written synchronously to SSD storage, but still only 23% of the "No log" throughput is achieved. Additionally, by employing specialized hardware, one arguably increases the costs and the management complexity of the nodes, especially in virtualized/cloud environments where such hardware may not be available in all machines.

There are works that avoid this penalty by using *asynchronous* writes to disk, allowing replicas to present a performance closer to the main memory system (e.g., Harp [28] and BFS [6]). The problem with this solution is that writing asynchronously does not give durability guarantees if all the replicas crash (and later recover), something that production systems need to address as correlated failures do happen [13, 15, 30, 33].

We would like to have a general solution that makes the performance of durable systems similar to pure memory systems, and that achieves this by *exploring the logging latency to process the requests* and by *optimizing log writes*.

Perturbations caused by checkpoints. Checkpoints are necessary to limit the log size, but their creation usually degrades the performance of the service. Figure 2 shows how the throughput of the SCKV-Store is affected by creating checkpoints at every 200K client requests. Taking a snapshot after processing a certain number of operations, as proposed in most works in SMR (e.g., [6, 27]), can make the system halt for a few seconds. This happens because requests are no longer processed while replicas save their state. Moreover, if the replicas are not fully synchronized, delays may also occur because the necessary agreement quorum might not be available.

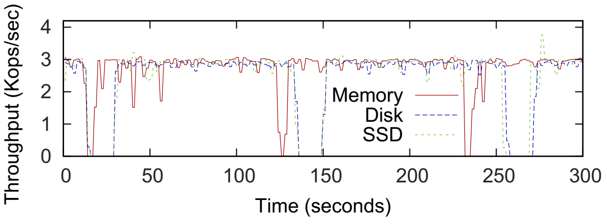


Figure 2: Throughput of a SCKV-Store with checkpoints in memory, disk and SSD considering a state of 1GB.

The figure indicates an equivalent performance degradation for checkpoints written in disk or SSD, meaning there is no extra benefit in using the latter (both require roughly the same amount of time to synchronously write the checkpoints). More importantly, the problem occurs even if the checkpoints are kept in memory, since the fundamental limitation is not due to storage accesses (as in logging), but to the cost to serialize a large state (1 GB).

Often, the performance decrease caused by checkpointing is not observed in the literature, either because no checkpoints were taken or because the service had a very small state (e.g., a counter with 8 bytes) [6, 10, 17, 21, 25]. Most of these works were focusing on ordering

[2]Synchronous writes are optimized to update only the file contents, and not the metadata, using the `rwd` mode in the Java' *RandomAccessFile* class (equivalent to using the `O_DSYNC` flag in POSIX *open*). This is important to avoid unnecessary disk head positioning.

requests efficiently, and therefore checkpointing could be disregarded as an orthogonal issue. Additionally, one could think that checkpoints need only to be created sporadically, and therefore, their impact is small on the overall execution. We argue that this is not true in many scenarios. For example, the SCKV-Store can process around 4700 4kB-writes per second (see §5), which means that the log can grow at the rate of more than 1.1 GB/min, and thus checkpoints need to be taken rather frequently to avoid outrageous log sizes. Leader-based protocols, such as those based on Paxos, have to log information about most of the exchanged messages, contributing to the log growth. Furthermore, recent SMR protocols require frequent checkpoints (every few hundred operations) to allow the service to recover efficiently from failed speculative request ordering attempts [17, 21, 25].

Some systems use *copy-on-write* techniques for doing checkpointing without stoping replicas (e.g., [9]), but this approach has two limitations. First, copy-on-write may be complicated to implement at application level in non-trivial services, as the service needs to keep track of which data objects were modified by the requests. Second, even if such techniques are employed, the creation of checkpoints still consumes resources and degrades the performance of the system. For example, writing a checkpoint to disk makes logging much slower since the disk head has to move between the log and checkpoint files, with the consequent disk seek times. In practice, this limitation could be addressed in part with extra hardware, such as by using two disks per server.

Another technique to deal with the problem is *fuzzy snapshots*, used in ZooKeeper [19]. A fuzzy snapshot is essentially a checkpoint that is done without stopping the execution of operations. The downside is that some operations may be executed more than once during recovery, an issue that ZooKeeper solves by forcing all operations to be idempotent. However, making operations idempotent requires non-trivial request pre-processing before they are ordered, and increases the difficulty of decoupling the replication library from the service [19, 20].

We aim to have a checkpointing mechanism that *minimizes performance degradation without requiring additional hardware and, at the same time, keeping the SMR programming model simple*.

Perturbations caused by state transfer. When a replica recovers, it needs to obtain an updated state to catch up with the other replicas. This state is usually composed of the last checkpoint plus the log up to some request defined by the recovering replica. Typically, (at least) another replica has to spend resources to send (part of) the state. If checkpoints and logs are stored in a disk, delays occur due to the transmission of the state through the network but also because of the disk accesses. Delta-checkpoint techniques based, for instance, on Merkle trees [6] can alleviate this problem, but cannot solve it completely since logs have always to be transferred. Moreover, implementing this kind of technique can add more complexity to the service code.

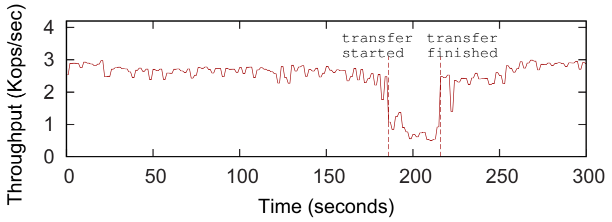


Figure 3: Throughput of a SCKV-Store when a failed replica recovers and asks for a state transfer.

Similarly to what is observed with checkpointing, there can be the temptation to disregard the state transfer impact on performance because it is perceived to occur rarely. However, techniques such as replica rejuvenation [18] and proactive recovery [6, 36] use state transfer to bring refreshed replicas up to date. Moreover, reconfigurations [29] and even leader change protocols (that need to be executed periodically for resilient BFT replication [10]) may require replicas to synchronize themselves [6, 35]. In conclusion, state transfer protocols may be invoked much more often than when there is a crash and a subsequent recovery.

Figure 3 illustrates the effect of state transmission during a replica recovery in a 4 node BFT system using the PBFT's state transfer protocol [6]. This protocol requires just one replica to send the state (checkpoint plus log) – similarly to crash FT Paxos-based systems – while others just provide authenticated hashes for state validation (as the sender of the state may suffer a Byzantine fault). The figure shows that the system performance drops to less than 1/3 of its normal performance during the 30 seconds required to complete state transfer. While one replica is recovering, another one is slowed because it is sending the state, and thus the remaining two are unable to order and execute requests (with $f = 1$, quorums of 3 replicas are needed to order requests).

One way to avoid this performance degradation is to ignore the state transfer requests until the load is low enough to process both the state transfers and normal request ordering [19]. However, this approach tends to delay the recovery of faulty replicas and makes the system vulnerable to extended unavailability periods (if more faults occur). Another possible solution is to add extra replicas to avoid interruptions on the service during recovery [36]. This solution is undesirable as it can increase the costs of deploying the system.

We would like to have a state transfer protocol that *minimizes the performance degradation due to state transfer without delaying the recovery of faulty replicas*.

3 Efficient Durability for SMR

In this section we present three techniques to solve the problems identified in the previous section.

3.1 Parallel Logging

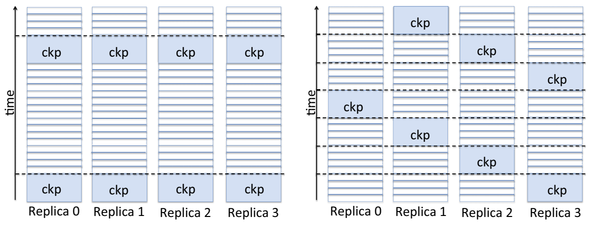
Parallel logging has the objective of hiding the high latency of logging. It is based on two ideas: (1) log groups of operations instead of single operations; and (2) process the operations in parallel with their storage.

The first idea explores the fact that disks have a high bandwidth, so the latency for writing 1 or 100 log entries can be similar, but the throughput would be naturally increased by a factor of roughly 100 in the second case. This technique requires the replication library to deliver groups of service operations (accumulated during the previous batch execution) to allow the whole batch to be logged at once, whereas previous solutions normally only provide single operations, one by one. Notice that this approach is different from the batching commonly used in SMR [6, 10, 25], where a group of operations is ordered together to amortize the costs of the agreement protocol (although many times these costs include logging a batch of requests to stable storage [27]). Here the aim is to pass batches of operations from the replication library to the service, and a batch may include (batches of) requests ordered in *different agreements*.

The second idea requires that the requests of a batch are processed while the corresponding log entries are being written to the secondary storage. Notice, however, that a reply can only be sent to the client after the corresponding request is executed *and logged*, ensuring that the result seen by the client will persist even if all replicas fail and later recover. Naturally, the effectiveness of this technique depends on the relation between the time for processing a batch and the time for logging it. More specifically, the interval T_k taken by a service to process a batch of k requests is given by $T_k = max(E_k, L_k)$, where E_k and L_k represent the latency of executing and logging the batch of k operations, respectively. This equation shows that the most expensive of the two operations (execution or logging) defines the delay for processing the batch. For example, in the case of the SCKV-Store, $E_k \ll L_k$ for any k, since inserting data in a hash table with chaining (an $\mathcal{O}(1)$ operation) is much faster than logging a 4kB-write (with or without batching). This is not the case for Durable DepSpace, which takes a much higher benefit from this technique (see §5).

3.2 Sequential Checkpointing

Sequential checkpointing aims at minimizing the performance impact of taking replica's state snapshots. The key principle is to exploit the natural redundancy that exists in asynchronous distributed systems based on SMR. Since these systems make progress as long as a quorum of $n - f$ replicas is available, there are f spare replicas in fault-free executions. The intuition here is to make each replica store its state at different times, to ensure that $n - f$ replicas can continue processing client requests.



(a) Synchronized. (b) Sequential.

Figure 4: Checkpointing strategies (4 replicas).

We define *global checkpointing period* P as the maximum number of (write) requests that a replica will execute before creating a new checkpoint. This parameter defines also the maximum size of a replica's log in number of requests. Although P is the same for all replicas, they checkpoint their state at different points of the execution. Moreover, all correct replicas will take at least one checkpoint within that period.

An instantiation of this model is for each replica $i = 0, ..., n-1$ to take a checkpoint after processing the k-th request where $k \bmod P = i \times \lfloor \frac{P}{n} \rfloor$, e.g., for $P = 1000$, $n = 4$, replica i takes a checkpoint after processing requests $i \times 250$, $1000 + i \times 250$, $2000 + i \times 250$, and so on.

Figure 4 compares a synchronous (or coordinated) checkpoint with our technique. Time grows from the bottom of the figure to the top. The shorter rectangles represent the logging of an operation, whereas the taller rectangles correspond to the creation of a checkpoint. It can be observed that synchronized checkpoints occur less frequently than sequential checkpoints, but they stop the system during their execution whereas for sequential checkpointing there is always an agreement quorum of 3 replicas available for continuing processing requests.

An important requirement of this scheme is to use values of P such that the chance of more than f overlapping checkpoints is negligible. Let C_{max} be the estimated maximum interval required for a replica to take a checkpoint and T_{max} the maximum throughput of the service. Two consecutive checkpoints will not overlap if:

$$\begin{aligned} C_{max} &< \frac{1}{T_{max}} \times \left\lfloor \frac{P}{n} \right\rfloor \implies \\ P &> n \times C_{max} \times T_{max} \qquad (1) \end{aligned}$$

Equation 1 defines the minimum value for P that can be used with sequential checkpoints. In our SCKV-Store example, for a state of 1GB and a 100% 4kB-write workload, we have $C_{max} \approx 15$s and $T_{max} \approx 4700$ ops/s, which means $P > 282000$. If more frequent checkpoints are required, the replicas can be organized in groups of at most f replicas to take checkpoints together.

3.3 Collaborative State Transfer

The state transfer protocol is used to update the state of a replica during recovery, by transmitting log records (L) and checkpoints (C) from other replicas (see Figure 5(a)). Typically only one of the replicas returns the full state and log, while the others may just send a hash of this data for validation (only required in the BFT case). As shown in §2, this approach can degrade performance during recoveries. Furthermore, it does not work with sequential checkpoints, as the received state can not be directly validated with hashes of other replicas' checkpoints (as they are different). These limitations are addressed with the *collaborative state transfer* (CST) protocol.

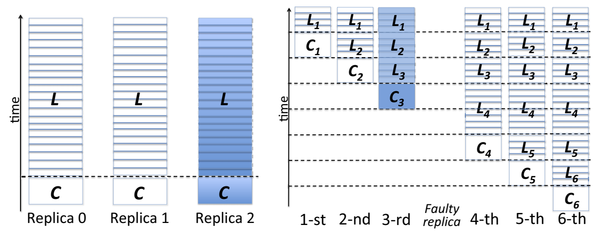
Although the two previous techniques work both with crash-tolerant and BFT SMR, the CST protocol is substantially more complex with Byzantine faults. Consequently, we start by describing a BFT version of the protocol (which also works for crash faults) and later, at the end of the section, we explain how CST can be simplified on a crash-tolerant system[3].

We designate by *leecher* the recovering replica and by *seeders* the replicas that send (parts of) their state. CST is triggered when a replica (leecher) starts (see Figure 6). Its first action is to use the local log and checkpoint to determine the last logged request and its sequence number (assigned by the ordering protocol), from now on called *agreement id*. The leecher then asks for the most recent logged agreement *id* of the other replicas, and waits for replies until $n - f$ of them are collected (including its own *id*). The *ids* are placed in a vector in descending order, and the largest *id* available in $f + 1$ replicas is selected, to ensure that such agreement *id* was logged by at least one correct replica (steps 1-3).

In BFT-SMaRt there is no parallel execution of agreements, so if one correct replica has ordered the *id*-th batch, it means with certainty that agreement *id* was already processed by at least $f + 1$ correct replicas[4]. The other correct replicas, which might be a bit late, will also eventually process this agreement, when they receive the necessary messages.

[3]Even though crash fault tolerance is by far more used in production systems, our choice is justified by two factors. First, the subtleties of BFT protocols require a more extensive discussion. Second, given the lack of a stable and widely-used open-source implementation of a crash fault tolerance SMR library, we choose to develop our techniques in a BFT SMR library, so the description is in accordance to our prototype.

[4]If one employs protocols such as Paxos/PBFT, low and high watermarks may need to considered.



(a) PBFT and others ($n = 4$). (b) CST ($n = 7$).

Figure 5: Data transfer in different state transfer strategies.

Next, the leecher proceeds to obtain the state up to *id* from a seeder and the associated validation data from f other replicas. The active replicas are ordered by the freshness of the checkpoints, from the most recent to the oldest (step 4). A leecher can make this calculation based on *id*, as replicas take checkpoints at deterministic points, as explained in §3.2. We call the replica with *i*-th oldest checkpoint the *i*-th replica and the checkpoint C_i. The log of a replica is divided in segments, and each segment L_i is the portion of the log required to update the state from C_i to the more recent state C_{i-1}. Therefore, we use the following notion of equivalence: $C_{i-1} \equiv C_i + L_i$. Notice that L_1 corresponds to the log records of the requests that were executed after the most recent checkpoint C_1 (see Figure 5(b) for $n = 7$).

The leecher fetches the state from the $(f + 1)$-th replica (seeder), which comprises the log segments L_1, ..., L_{f+1} and checkpoint C_{f+1} (step 8). To validate this state, it also gets hashes of the log segments and checkpoints from the other f replicas with more recent checkpoints (from the 1st until the f-th replica) (step 6a). Then, the leecher sets its state to the checkpoint and replays the log segments received from the seeder, in order to bring up to date its state (steps 10 and 12a).

The state validation is performed by comparing the hashes of the f replicas with the hashes of the log segments from the seeder and intermediate checkpoints. For each replica i, the leecher replays L_{i+1} to reach a state equivalent to the checkpoint of this replica. Then, it creates a intermediate checkpoint of its state and calculates the corresponding hash (steps 12a and 12b). The leecher finds out if the log segments sent by the seeder and the current state (after executing L_{i+1}) match the hashes provided by this replica (step 12c).

If the check succeeds for f replicas, the reached state is valid and the CST protocol can finish (step 13). If the validation fails, the leecher fetches the data from the $(f + 2)$-th replica, which includes the log segments L_1, ..., L_{f+2} and checkpoint C_{f+2} (step 13 goes back to step 8). Then, it re-executes the validation protocol, considering as extra validation information the hashes that were produced with the data from the $(f + 1)$-th replica (step 9). Notice that the validation still requires $f + 1$ matching

```
1. Look at the local log to discover the last executed agreement;
2. Fetch the id of the last executed agreement from $n-f$ replicas
   (including itself) and save the identifier of these replicas;
3. $id$ = largest agreement id that is available in $f+1$ replicas;
4. Using $id$, $P$ and $n$, order the replicas (including itself) with the
   ones with most recent checkpoints first;
5. $V \leftarrow \emptyset$; // the set containing state and log hashes
6. For $i = 1$ to $f$ do:
   (a) Fetch $V_i = \langle HL_1, ..., HL_i, HC_i \rangle$ from $i$-th replica;
   (b) $V \leftarrow V \cup \{V_i\}$;
7. $r \leftarrow f+1$; // replica to fetch state
8. Fetch $S_r = \langle L_1, ..., L_r, C_r \rangle$ from $r$-th replica;
9. $V \leftarrow V \cup \{\langle \mathsf{H}(S_r.L_1), ..., \mathsf{H}(S_r.L_r), \mathsf{H}(S_r.C_r) \rangle\}$;
10. Update state using $S_r.C_r$;
11. $v \leftarrow 0$; // number of validations of $S_r$
12. For $i = r-1$ down to 1 do:
    (a) Replay log $S_r.L_{i+1}$;
    (b) Take checkpoint $C'_i$ and calculate its hash $HC'_i$;
    (c) If $(V_i.HL_{1..i} = V_r.HL_{1..i}) \wedge (V_i.HC_i = HC'_i)$, $v++$;
13. If $v \geq f$, replay log $S_r.L_1$ and return; Else, $r++$ and go to 8;
```

Figure 6: The CST recovery protocol called by the leecher after a restart. *Fetch* commands wait for replies within a timeout and go back to step 2 if they do not complete.

log segments and checkpoints, but now there are $f+2$ replicas involved, and the validation is successful even with one Byzantine replica. In the worst case, f faulty replicas participate in the protocol, which requires $2f+1$ replicas to send some data, ensuring a correct majority and at least one valid state (log and checkpoint).

In the scenario of Figure 5(b), the 3rd replica (the $(f+1)$-th replica) sends L_1, L_2, L_3 and C_3, while the 2nd replica only transmits $HL_1 = \mathsf{H}(L_1)$, $HL_2 = \mathsf{H}(L_2)$ and $HC_2 = \mathsf{H}(C_2)$, and the 1st replica sends $HL_1 = \mathsf{H}(L_1)$ and $HC_1 = \mathsf{H}(C_1)$. The leecher next replays L_3 to get to state $C_3 + L_3$, and takes the intermediate checkpoint C'_2 and calculates the hash $HC'_2 = \mathsf{H}(C'_2)$. If HC'_2 matches HC_2 from the 2nd replica, and the hashes of log segments L_2 and L_1 from the 3rd replica are equal to HL_2 and HL_1 from the 2nd replica, then the first validation is successful. Next, a similar procedure is applied to replay L_2 and the validation data from the 1st replica. Now, the leecher only needs to replay L_1 to reach the state corresponding to the execution of request id.

While the state transfer protocol is running, replicas continue to create new checkpoints and logs since the recovery does not stop the processing of new requests. Therefore, they are required to keep old log segments and checkpoints to improve their chances to support the recovery of a slow leecher. However, to bound the required storage space, these old files are eventually removed, and the leecher might not be able to collect enough data to complete recovery. When this happens, it restarts the algorithm using a more recent request *id* (a similar solution exists in all other state state transfer protocols that we are aware of, e.g., [6, 8]).

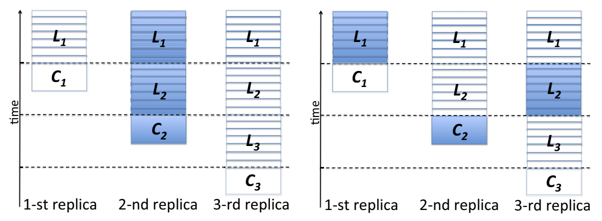
The leecher observes the execution of the other replicas while running CST, and stores all received messages concerning agreements more recent than *id* in an out-of-context buffer. At the end of CST, it uses this buffer to catch up with the other replicas, allowing it to be reintegrated in the state machine.

Correctness. We present here a brief correctness argument of the CST protocol. Assume that b is the actual number of faulty (Byzantine) replicas (lower or equal to f) and r the number of recovering replicas.

In terms of safety, the first thing to observe is that CST returns if and only if the state is validated by at least $f+1$ replicas. This implies that the state reached by the leecher at the end of the procedure is valid according to at least one correct replica. To ensure that this state is recent, the largest agreement *id* that is returned by $f+1$ replicas is used.

Regarding liveness, there are two cases to consider. If $b+r \leq f$, there are still $n-f$ correct replicas running and therefore the system could have made progress while the r replicas were crashed. A replica is able to recover as long as checkpoints and logs can be collected from the other replicas. Blocking is prevented because CST restarts if any of the *Fetch* commands fails or takes too much time. Consequently, the protocol is live if correct replicas keep the logs and checkpoints for a sufficiently long interval. This is a common assumption for state transfer protocols. If $b+r > f$, then there may not be enough replicas for the system to continue processing. In this case the recovering replica(s) will continuously try to fetch the most up to date agreement *id* from $n-f$ replicas (possibly including other recovering replicas) until such quorum exists. Notice that a total system crash is a special case of this scenario.

Optimizing CST for $f = 1$. When $f = 1$ (and thus $n = 4$), a single recovering replica can degrade the performance of the system because one of $n-f$ replicas will be transferring the checkpoint and logs, delaying the execution of the agreements (as illustrated in Figure 7(a)). To avoid this problem, we spread the data transfer between the active replicas through the following optimization in an *initial recovery round*: the 2nd replica ($f+1 = 2$) sends C_2 plus $\langle HL_1, HL_2 \rangle$ (instead of the checkpoint plus full log), while the 1st replica sends L_1 and HC_1 (instead of only hashes) and the 3rd replica sends L_2 (instead of not participating). If the validation of the received state



(a) General CST. (b) Optimized CST.

Figure 7: General and optimized CST with $f = 1$.

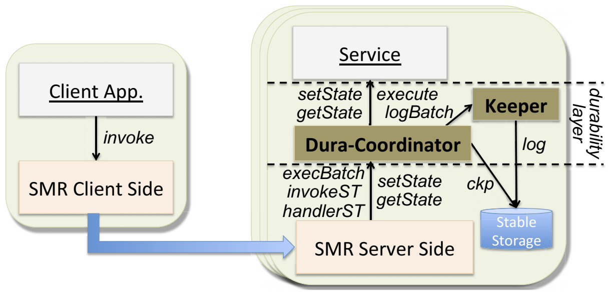


Figure 8: The Dura-SMaRt architecture.

fails, then the normal CST protocol is executed. This optimization is represented in Figure 7(b), and in §5 we show the benefits of this strategy.

Simplifications for crash faults. When the SMR only needs to tolerate crash faults, a much simpler version of CST can be employed. The basic idea is to execute steps 1-4 of CST and then fetch and use the checkpoint and log from the 1st (most up to date) replica, since no validation needs to be performed. If $f = 1$, a analogous optimization can be used to spread the burden of data transfer among the two replicas: the 1st replica sends the checkpoint while the 2nd replica sends the log segment.

4 Implementation: Dura-SMaRt

In order to validate our techniques, we extended the open-source BFT-SMaRt replication library [1] with a durability layer, placed between the request ordering and the service. We named the resulting system *Dura-SMaRt*, and used it to implement two applications: a consistent key-value store and a coordination service.

Adding durability to BFT-SMaRt. BFT-SMaRt originally offered an API for invoking and executing state machine operations, and some callback operations to fetch and set the service state. The implemented protocols are described in [35] and follow the basic ideas introduced in PBFT and Aardvark [6, 10]. BFT-SMaRt is capable of ordering more than 100K 0-byte msg/s (the 0/0 microbenchmark used to evaluate BFT protocols [17, 25]) in our environment. However, this throughput drops to 20K and 5K msgs/s for 1kB and 4kB message sizes, respectively (the workloads we use – see §5).

We modified BFT-SMaRt to accommodate an intermediate *Durability layer* implementing our techniques at the server-side, as described in Figure 8, together with the following modifications on BFT-SMaRt. First, we added a new server side operation to deliver batches of requests instead of one by one. This operation supplies ordered but not delivered requests spanning one or more agreements, so they can be logged in a single write by the *Keeper* thread. Second, we implemented the parallel checkpoints and collaborative state transfer in the *Dura-Coordinator* component, removing the old checkpoint and state transfer logic from BFT-SMaRt and defining an extensible API for implementing different state transfer strategies. Finally, we created a dedicated thread and socket to be used for state transfer in order to decrease its interference on request processing.

SCKV-store. The first system implemented with Dura-SMaRt was a *simple and consistent key-value store* (SCKV-Store) that supports the storage and retrieval of key-value pairs, alike to other services described in the literature, e.g., [11, 31]. The implementation of the SCKV-Store was greatly simplified, since consistency and availability come directly from SMR and durability is achieved with our new layer.

Durable DepSpace (DDS). The second use case is a durable extension of the DepSpace coordination service [2], which originally stored all data only in memory. The system, named Durable DepSpace (DDS), provides a tuple space interface in which tuples (variable-size sequences of typed fields) can be inserted, retrieved and removed. There are two important characteristics of DDS that differentiate it from similar services such as Chubby [4] and ZooKepper [19]: it does not follow a hierarchical data model, since tuple spaces are, by definition, unstructured; and it tolerates Byzantine faults, instead of only crash faults. The addition of durability to DepSpace basically required the replacement of its original replication layer by Dura-SMaRt.

5 Evaluation

This section evaluates the effectiveness of our techniques for implementing durable SMR services. In particular, we devised experiments to answer the following questions: (1) What is the cost of adding durability to SMR services? (2) How much does *parallel logging* improve the efficiency of durable SMR with synchronous disk and SSD writes? (3) Can *sequential checkpoints* remove the costs of taking checkpoints in durable SMR? (4) How does *collaborative state transfer* affect replica recoveries for different values of f? Question 1 was addressed in §2, so we focus on questions 2-4.

Case studies and workloads. As already mentioned, we consider two SMR-based services implemented using Dura-SMaRt: the SCKV-Store and the DDS coordination service. Although in practice, these systems tend to serve mixed or read-intensive workloads [11, 19], we focus on write operations because they stress both the ordering protocol and the durable storage (disk or SSD). Reads, on the other hand, can be served from memory, without running the ordering protocol. Therefore, we consider a 100%-write workload, which has to be processed by an agreement, execution and logging. For the SCKV-Store, we use YCSB [11] with a new workload composed of 100% of replaces of 4kB-values, making our results comparable to other recent SMR-based storage systems [3, 32, 37]. For DDS, we consider the insertion of random tuples with four fields containing strings, with a total size of 1kB, creating a workload with a pattern equivalent to the ZooKeeper evaluation [19, 20].

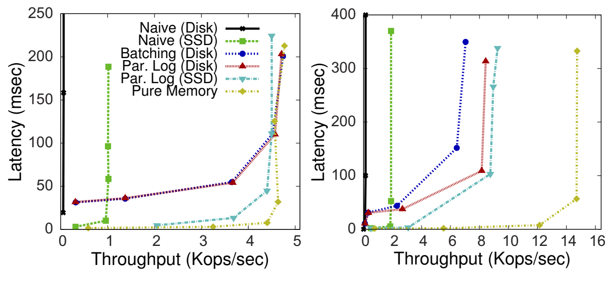
Experimental environment. All experiments, including the ones in §2, were executed in a cluster of 14 machines interconnected by a gigabit ethernet. Each machine has two quad-core 2.27 GHz Intel Xeon E5520, 32 GB of RAM memory, a 146 GB 15000 RPM SCSI disk and a 120 GB SATA Flash SSD. We ran the IOzone benchmark[5] on our disk and SSD to understand their performance under the kind of workload we are interested: rewrite (append) for records of 1MB and 4MB (the maximum size of the request batch to be logged in DDS and SCKV-Store, respectively). The results are:

Record length	Disk	SSD
1MB	96.1 MB/s	128.3 MB/s
4MB	135.6 MB/s	130.7 MB/s

Parallel logging. Figure 9(a) displays latency-throughput curves for the SCKV-Store considering several durability variants. The figure shows that naive (synchronous) disk and SSD logging achieve a throughput of 63 and 1017 ops/s, respectively, while a pure memory version with no durability reaches a throughput of around 4772 ops/s.

Parallel logging involves two ideas, the storage of batches of operations in a single write and the execution of operations in parallel with the secondary storage accesses. The use of batch delivery alone allowed for a throughput of 4739 ops/s with disks (a 75× improvement over naive disk logging). This roughly represents what would be achieved in Paxos [24, 27], ZooKeeper [19]

[5]http://www.iozone.org.



(a) SCKV-Store. (b) Durable DepSpace.

Figure 9: Latency-throughput curves for several variants of the SCKV-Store and DDS considering 100%-write workloads of 4kB and 1kB, respectively. Disk and SSD logging are always done synchronously. The legend in (a) is valid also for (b).

or UpRight [9], with requests being logged during the agreement protocol. Interestingly, the addition of a separated thread to write the batch of operations, does not improve the throughput of this system. This occurs because a local put on SCKV-Store replica is very efficient, with almost no effect on the throughput.

The use of parallel logging with SSDs improves the latency of the system by 30-50ms when compared with disks until a load of 4400 ops/s. After this point, parallel logging with SSDs achieves a peak throughput of 4500 ops/s, 5% less than parallel logging with disk (4710 ops/s), with the same observed latency. This is consistent with the IOzone results. Overall, parallel logging with disk achieves 98% of the throughput of the pure memory solution, being the replication layer the main bottleneck of the system. Moreover, the use of SSDs neither solves the problem that parallel logging addresses, nor improves the performance of our technique, being thus not effective in eliminating the log bottleneck of durable SMR.

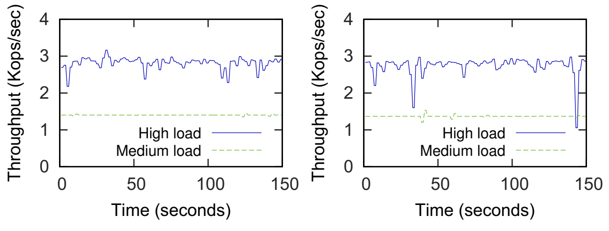
Figure 9(b) presents the results of a similar experiment, but now considering DDS with the same durability variants as in SCKV-Store. The figure shows that a version of DDS with naive logging in disk (resp. SSD) achieves a throughput of 143 ops/s (resp. 1900 ops/s), while a pure memory system (DepSpace), reaches 14739 ops/s. The use of batch delivery improves the performance of disk logging to 7153 ops/s (a 50× improvement). However, differently from what happens with SCKV-Store, the use of parallel logging in disk further improves the system throughput to 8430 ops/s, an improvement of 18% when compared with batching alone. This difference is due to the fact that inserting a tuple requires traversing many layers [2] and the update of an hierarchical index, which takes a non-negligible time (0.04 ms), and impacts the performance of the system if done sequentially with logging. The difference would be even bigger if the SMR service requires more processing. Finally, the use of SSDs with parallel logging in DDS was more effective than with the SCKV-Store, increasing the peak throughput of the system to 9250 ops/s (an improvement of 10% when compared with disks). Again, this is consistent with our IOzone results: we use 1kB requests here, so the batches are smaller than in SCKV-Store, and SSDs are more efficient with smaller writes.

Notice that DDS could not achieve a throughput near to pure memory. This happens because, as discussed in §3.1, the throughput of parallel logging will be closer to a pure memory system if the time required to process a batch of requests is akin to the time to log this batch. In the experiments, we observed that the workload makes BFT-SMaRt deliver batches of approximately 750 requests on average. The local execution of such batch takes around 30 ms, and the logging of this batch on disk entails 70 ms. This implies a maximum throughput of 10.750 ops/s, which is close to the obtained values. With this workload, the execution time matches the log time (around 500 ms) for batches of 30K operations. These batches require the replication library to reach a throughput of 60K 1kB msgs/s, three times more than what BFT-SMaRt achieves for this message size.

Sequential Checkpointing. Figure 10 illustrates the effect of executing sequential checkpoints in disks with SCKV-Store[6] during a 3-minute execution period.

When compared with the results of Figure 2 for synchronized checkpoints, one can observe that the unavailability periods no longer occur, as the 4 replicas take checkpoints separately. This is valid both when there is a high and medium load on the service and with disks and SSDs (not show). However, if the system is under stress (high load), it is possible to notice a periodic small decrease on the throughput happening with both 500MB and 1GB states (Figures 10(a) and 10(b)). This behavior is justified because at every $\lfloor \frac{P}{n} \rfloor$ requests one of the replicas takes a checkpoint. When this occurs, the replica stops executing the agreements, which causes it to become a bit late (once it resumes processing) when compared with the other replicas. While the replica is still catching up, another replica initiates the checkpoint, and therefore, a few agreements get delayed as the quorum is not immediately available. Notice that this effect does not exist if the system has less load or if there is sufficient time between sequential checkpoints to allow replicas to catch up ("Medium load" line in Figure 10).

[6]Although we do not show checkpoint and state transfer results for DDS due to space constraints, the use of our techniques bring the same advantage as on SCKV-Store. The only noticeable difference is due to the fact that DDS local tuple insertions are more costly than SCKV-Store local puts, which makes the variance on the throughput of sequential checkpoints even more noticeable (especially when the leader is taking its checkpoint). However, as in SCKV-Store, this effect is directly proportional to the load imposed to the system.



(a) 500MB state. (b) 1GB state.

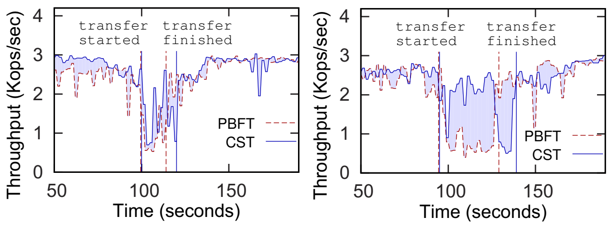
Figure 10: SCKV-Store throughput with sequential checkpoints with different write-only loads and state size.

Collaborative State Transfer. This section evaluates the benefits of CST when compared to a PBFT-like state transfer in the SCKV-Store with disks, with 4 and 7 replicas, considering two state sizes. In all experiments a single replica recovery is triggered when the log size is approximately twice the state size, to simulate the condition of Figure 7(b).

Figure 11 displays the observed throughput of some executions of a system with $n = 4$, running PBFT and the CST algorithm optimized for $f = 1$, for states of 500MB and 1GB, respectively. A PBFT-like state transfer takes 30 (resp. 16) seconds to deliver the whole 1 GB (resp. 500MB) of state with a sole replica transmitter. In this period, the system processes 741 (resp. 984) write ops/sec on average. CST optimized for $f = 1$ divides the state transfer by three replicas, where one sends the state and the other two up to half the log each. Overall, this operation takes 42 (resp. 20) seconds for a state of 1GB (resp. 500MB), 28% (resp. 20%) more than with the PBFT-like solution for the same state size. However, during this period the system processes 1809 (resp. 1426) ops/sec on average. Overall, the SCKV-Store with a state of 1GB achieves only 24% (or 32% for 500MB-state) of its normal throughput with a PBFT-like state transfer, while the use of CST raises this number to 60% (or 47% for 500MB-state).

Two observations can be made about this experiment. First, the benefit of CST might not be as good as expected for small states (47% of the normal throughput for a 500MB-state) due to the fact that when fetching state from different replicas we need to wait for the slowest one, which always brings some degradation in terms of time to fetch the state (20% more time). Second, when the state is bigger (1GB), the benefits of dividing the load among several replicas make state transfer much less damaging to the overall system throughput (60% of the normal throughput), even considering the extra time required for fetching the state (+28%).

We did an analogous experiment for $n = 7$ (not shown due to space constraints) and observed that, as expected, the state transfer no longer causes a degradation on the system throughput (both for CST and PBFT) since state is fetched from a single replica, which is available since



(a) 500MB and $n = 4$. (b) 1GB and $n = 4$.

Figure 11: Effect of a replica recovery on SCKV-Store throughput using CST with $f = 1$ and different state sizes.

$n = 7$ and there is only one faulty replica (see Figure 5). We repeated the experiment for $n = 7$ with the state of 1GB being fetched from the leader, and we noticed a 65% degradation on the throughput. A comparable effect occurs if the state is obtained from the leader in CST. As a cautionary note, we would like to remark that when using spare replicas for "cheap" faulty recovery, it is important to avoid fetching the state from the leader replica (as in [4, 8, 19, 32]) because this replica dictates the overall system performance.

6 Related Work

Over the years, there has been a reasonable amount of work about stable state management in main memory databases (see [16] for an early survey). In particular, parallel logging shares some ideas with classical techniques such as group commit and pre-committed transactions [14] and the creation of checkpoints in background has also been suggested [26]. Our techniques were however developed with the SMR model in mind, and therefore, they leverage the specific characteristics of these systems (e.g., log groups of requests while they are executed, and schedule checkpoints preserving the agreement quorums).

Durability management is a key aspect of practical crash-FT SMR-like systems [3, 8, 19, 20, 32, 37]. In particular, making the system use the disk efficiently usually requires several hacks and tricks (e.g., non-transparent copy-on-write, request throttling) on an otherwise small and simple protocol and service specification [8]. These systems usually resort to dedicated disks for logging, employ mostly synchronized checkpoints and fetch the state from a leader [8, 19, 32]. A few systems also delay state transfer during load-intensive periods to avoid a noticeable service degradation [19, 37]. All these approaches either hurt the SMR elegant programming model or lead to the problems described in §2.2. For instance, recent consistent storage systems such as Windows Azure Storage [5] and Spanner [12] use Paxos together with several extensions for ensuring durability. We believe works like ours can improve the modularity of future systems requiring durable SMR techniques.

BFT SMR systems use logging, checkpoints, and state transfer, but the associated performance penalties often do not appear in the papers because the state is very small (e.g., a counter) or the checkpoint period is too large (e.g., [6, 10, 17, 21, 25]). A notable exception is UpRight [9], which implements durable state machine replication, albeit without focusing on the efficiency of logging, checkpoints and state transfer. In any case, if one wants to sustain a high-throughput (as reported in the papers) for non-trivial states, the use of our techniques is fundamental. Moreover, any implementation of proactive recovery [6, 36] requires an efficient state transfer.

PBFT [6] was one of the few works that explicitly dealt with the problem of optimizing checkpoints and state transfer. The proposed mechanism was based on copy-on-write and delta-checkpoints to ensure that only pages modified since the previous checkpoint are stored. This mechanism is complementary to our techniques, as we could use it together with the sequential checkpoints and also to fetch checkpoint pages in parallel from different replicas to improve the state transfer. However, the use of copy-on-write may require the service definition to follow certain abstractions [7, 9], which can increase the complexity of the programming model. Additionally, this mechanism, which is referred in many subsequent works (e.g., [17, 25]), only alleviates but does not solve the problems discussed in §2.2.

A few works have described solutions for fetching different portions of a database state from several "donors" for fast replica recovery or database cluster reconfiguration (e.g., [23]). The same kind of techniques were employed for fast replica recovery in group communication systems [22] and, more recently, in main-memory-based storage [31]. There are three differences between these works and ours. First, these systems try to improve the recovery time of faulty replicas, while CST main objective is to minimize the effect of replica recovery on the system performance. Second, we are concerned with the interplay between logging and checkpoints, which is fundamental in SMR, while these works are more concerned with state snapshots. Finally, our work has a broader scope in the sense that it includes a set of complementary techniques for Byzantine and crash faults in SMR systems, while previous works address only crash faults.

7 Conclusion

This paper discusses several performance problems caused by the use of logging, checkpoints and state transfer on SMR systems, and proposes a set of techniques to mitigate them. The techniques – parallel logging, sequential checkpoints and collaborative state transfer – are purely algorithmic, and require no additional support (e.g., hardware) to be implemented in commodity servers. Moreover, they preserve the simple state machine programming model, and thus can be integrated in any crash or Byzantine fault-tolerant library without impact on the supported services.

The techniques were implemented in a durability layer for the BFT-SMaRt library, which was used to develop two representative services: a KV-store and a coordination service. Our results show that these services can reach up to 98% of the throughput of pure memory systems, remove most of the negative effects of checkpoints and substantially decrease the throughput degradation during state transfer. We also show that the identified performance problems can not be solved by exchanging disks by SSDs, highlighting the need for techniques such as the ones presented here.

Acknowledgements. Thanks to the anonymous reviewers, John Howell and Lorenzo Alvisi, our shepherd, for the comments that helped improve the paper. This work was partially supported by the EC FP7 through project TCLOUDS (ICT-257243), by the FCT through project RC-Clouds (PTDC/EIA-EIA/115211/2009), the Multi-annual Program (LASIGE), and contract PEst-OE/EEI/LA0021/2011 (INESC-ID).

References

[1] BFT-SMaRt project page. `http://code.google.com/p/bftsmart`, 2012.

[2] A. Bessani, E. Alchieri, M. Correia, and J. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *EuroSys*, 2008.

[3] W. Bolosky, D. Bradshaw, R. Haagens, N. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *NSDI*, 2011.

[4] M. Burrows. The Chubby lock service. In *OSDI*, 2006.

[5] B. Calder et al. Windows azure storage: A highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[6] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[7] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21(3):236–269, Aug. 2003.

[8] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live - An engineering perspective. In *PODC*, 2007.

[9] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché. UpRight cluster services. In *SOSP*, 2009.

[10] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, 2009.

[11] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SOCC*, 2010.

[12] J. Corbett et al. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[13] J. Dean. Google: Designs, lessons and advice from building large distributed systems. In *Keynote at LADIS*, Oct. 2009.

[14] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, 1984.

[15] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *OSDI*, 2010.

[16] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, Dec. 1992.

[17] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *EuroSys*, 2010.

[18] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: analysis, module and applications. In *FTCS*, 1995.

[19] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for Internet-scale services. In *USENIX ATC*, 2010.

[20] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, 2011.

[21] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. In *EuroSys*, 2012.

[22] R. Kapitza, T. Zeman, F. Hauck, and H. P. Reiser. Parallel state transfer in object replication systems. In *DAIS*, 2007.

[23] B. Kemme, A. Bartoli, and O. Babaoglu. Online reconfiguration in replicated databases based on group communication. In *DSN*, 2001.

[24] J. Kirsh and Y. Amir. Paxos for system builders: An overview. In *LADIS*, 2008.

[25] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39, Dec. 2009.

[26] K.-Y. Lam. An implementation for small databases with high availability. *SIGOPS Operating Systems Rev.*, 25(4), Oct. 1991.

[27] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[28] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the Harp file system. In *SOSP*, 1991.

[29] J. Lorch, A. Adya, W. Bolosky, R. Chaiken, J. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. In *EuroSys*, 2006.

[30] R. Miller. Explosion at The Planet causes major outage. *Data Center Knowledge*, June 2008.

[31] D. Ongaro, S. M. Ruble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

[32] J. Rao, E. J. Shenkita, and S. Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *VLDB*, 2011.

[33] M. Ricknäs. Lightning strike in Dublin downs Amazon, Microsoft clouds. *PC World*, Aug. 2011.

[34] F. B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[35] J. Sousa and A. Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *EDCC*, 2012.

[36] P. Sousa, A. Bessani, M. Correia, N. Neves, and P. Veríssimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, Apr. 2010.

[37] Y. Wang, L. Alvisi, and M. Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *USENIX ATC*, 2012.