Elastic State Machine Replication

Andre Nogueira, Antonio Casimiro, Alysson Bessani

State machine replication (SMR) is a fundamental technique for implementing stateful dependable systems. A key limitation of this technique is that the performance of a service does not scale with the number of replicas hosting it. Some works have shown that such scalability can be achieved by partitioning the state of the service into shards. The few SMR-based systems that support dynamic partitioning implement ad-hoc state transfer protocols and perform scaling operations as background tasks to minimize the performance degradation during reconfigurations. In this work we go one step further and propose a modular partition transfer protocol for creating and destroying such partitions at runtime, thus providing fast elasticity for crash and Byzantine fault tolerant replicated state machines and making them more suitable for cloud systems.

Index Terms—State machine replication, replication, elasticity, fault tolerance, Byzantine fault tolerance, partitioning, scaling.

1 Introduction

State Machine Replication (SMR) is a well-known approach to replicate a service for fault tolerance [55]. The key idea is to make replicas deterministically execute the same sequence of requests in such a way that, despite the failure of a fraction of the replicas, the remaining ones have the same state and ensure the availability of the system. Many production systems use this approach to tolerate crash faults [2], [3], [13], [12], [15], [18], [30], mostly by implementing Paxos [37] or a similar algorithm [31], [49], [50].

A critical limitation of the basic SMR approach is its lack of scalability: (1) the services usually need to be single-threaded to ensure replica determinism [55], (2) there is normally a leader replica which is the bottleneck of the SMR ordering protocol, and (3) adding more replicas does not improve system performance. Different techniques have been developed to deal with these limitations. Some works propose SMR implementations that take advantage of multiple cores to execute requests in parallel [32], [34], [43], solving (1). Although effective, the improvements are limited by the number of cores available on servers. In the same way, some protocols spread the additional load imposed on the leader among all the system replicas [42], [44]. These protocols solve (2), but scalability remains limited because every replica still needs to execute all the operations.

A recent line of work proposes partitioning of the SMR-based systems in multiple Replicated State Machines (RSMs) [10], [18], [27], [38], [51] for addressing (3). Although partitioning solves the *scalability of SMR*, the existing solutions are quite limited in terms of *elasticity*, i.e., the capacity to dynamically increase (scale-out) and decrease (scale-in) the number of partitions at runtime. More specifically, some scalable systems consider only static partitions [10], [38], [51], with no elasticity at all, while others [18], [27] provide dynamic partitioning through ad-hoc protocols that are executed in the background to avoid performance disruption, but with negative implications on the time needed to complete the partitioning.

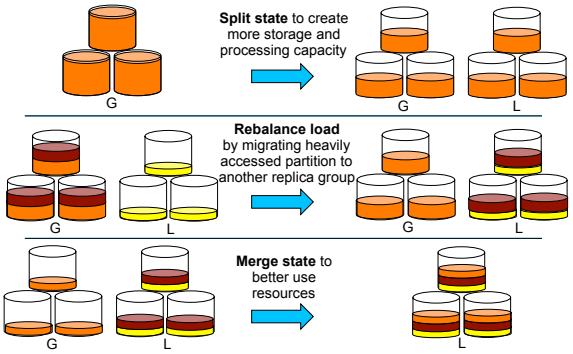


Figure 1. Reconfigurations of a partitionable RSM.

This paper introduces a generic *partition transfer* primitive (and protocol) that enables SMR-based services to be *elastic*. The proposed protocol is designed to perform partition transfers efficiently and with minimal perturbations on the client-perceived performance on top of any existing SMR protocol.

SMR is considered the main technique for maintaining critical state in a distributed system [48]. Although it was traditionally employed for building metadata and coordination services (e.g., [13], [12], [30]), recent works have been pushing the use of this technique for implementing high-performance storage services [2], [3], [8], [11], [18], [51], [52], [60]. *Elastic SMR* gives these services the ability to increase and decrease their capacity in terms of storage and throughput. Figure 1 illustrates three situations in which such elasticity might be beneficial. A group G is used up to its maximum load capacity and, then, the state managed by this group is *split* to another group L to balance the load among the two groups and open room for the system to handle more work. When two groups are processing requests, a non-uniform access pattern might unbalance the load handled by the two groups, as shown in the second case. Here, part of the state of the overloaded group G can be transferred to the underloaded group L to *rebalance* the system. Finally, if the load decreases and one of the partitions becomes underutilized, it is possible to

- *The authors are with LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal.*
- *This work was supported by FCT through projects LaSIGE (UID/CEC/00408/2013) and IRCoC (PTDC/EEI-SCR/6970/2014), and by the European Commission through the H2020 programme under grant agreement 643964 (SUPERCLOUD project).*

merge the state of the two groups in a single RSM, for freeing the underutilized resources.

An important issue in elastic systems is to define when and how to perform reconfigurations like the ones shown in Figure 1. There are many works on dynamic resource configuration managers (e.g., [26], [35], [46]), which decide when and how to adapt according to specified policies. A fundamental parameter used in these systems is the amount of time required for deploying a new server and making it ready to process requests, i.e., the *setup cost*. A high setup cost always discourages reconfigurations, leading to over-provisioning and increased operational costs [26]. Given that, it is extremely important to reduce the setup cost.

In contrast to a stateless service, where servers start processing requests as soon as they are launched, in a stateful service a partition transfer must be performed before a server starts processing requests. As a result, the setup cost increases. Therefore, tackling the cost of performing a partition transfer is crucial when stateful services are deployed on cloud systems in order to meet the requirements defined in Service Level Agreements (SLA). Usually, elastic stateful systems rely on distributed cache layers [47], write-offloading [45] and, ultimately, over-provisioning [26]. These techniques are useful to create buffers to either absorb unexpected load spikes or buy time for the stateful backend to reconfigure. In this work we challenge this design, at least for an important class of systems (SMR- or Paxos-based systems), and define a principled way for replicated stateful systems to scale in and out efficiently, for non-negligible partition sizes and workloads.

We implement our partition transfer protocol in an existing state machine replication programming library, and build a strongly-consistent elastic key-value store on top of it. We evaluate this system by performing several scaling operations, measuring the duration of partition transfers and the impact of these reconfigurations on the latency and throughput of the system. Our results show that the proposed solution effectively supports *fast reconfigurations*, allowing the system to quickly adapt to changes in its workload, something that is not achievable in current elastic (stateful) systems [17], [19], [33].

In summary, the contributions of this paper are the following:

1) It surveys how elasticity is (or can be) implemented in existing SMR-based services and experimentally demonstrates their inefficiencies (§2);
2) It introduces a modular partition transfer primitive and protocol that enables SMR-based services to efficiently split and merge their state (§3);
3) It describes an implementation of this primitive in an open-source SMR library (§4);
4) It provides an extensive evaluation of the partition transfer protocol assessing its impact on key SLA-related metrics such as the duration of the reconfiguration process, the service throughput, and the operation latency (§5).

2 Elasticity for RSMs

Dividing a RSM in multiple independent shards allows the system to scale linearly with the number of servers (as long as most operations affect a single partition). Most works following this approach assume statically defined partitions. The few that do support dynamic state partitioning neither specify how the partition is transferred to the new set of replicas nor how to make such transference as fast as possible and with minimal performance interference on the system. Having a *partition transfer primitive* for executing the operations illustrated in Figure 1 will allow SMR-based services to grow and shrink with the demand, thus supporting elasticity. In this section we discuss some potential solutions for implementing such a primitive and their limitations.

2.1 Partition transfer in existing RSMs

None of the classical SMR protocols support a primitive for dynamically transferring partitions. However, this functionality can be added to an existing SMR-based service with minor or no modifications to the replication library. In the following we describe and evaluate two simple solutions that can be easily integrated in existing protocols. The objective is to characterize a baseline performance, which can be obtained with such straightforward approaches.

2.1.1 A client-based solution

Our first candidate solution can be integrated to existing systems by adding three new operations to the service implemented as a RSM, but without modifying the replication library. As shown in Figure 2(a), the idea is to have a special client (coordinator) that moves part of the state from a source RSM to a destination RSM, which will host the partition. To ensure there is no violation of the consistency of the service, the source RSM stops serving the transferred partition right after the first step, although this data is only deleted after it is installed in the destination RSM.

Unfortunately, this straightforward design does not work well for large partitions. To show that, we implemented this protocol for a simple and consistent key-value store built on top of BFT-SMaRt[1] [9] and conducted some experiments. We used the YCSB benchmark read-heavy workload (95% reads and 5% writes) [17] to measure the performance of the system during a 4GB-partition transfer. Figures 2(b) and 2(c) show the throughput and the 99-percentile latency, calculated at 2-second intervals (see details about our experimental environment and methodology in §5), when the protocol described above is used for transferring 1024 blocks of 4MB and 256 blocks of 16MB, respectively.

The figures show that transferring a 4GB-partition takes 16.3 minutes, almost 22× more than a 4GB-transfer between two machines using *rsync* (see §5.3). More importantly, client operations accessing the block being transferred are blocked until the operation completes, causing spikes on the latency. This effect is more prominent when larger key-value blocks are transferred (Figure 2(c)), as the throughput decreases and latency spikes occur. When we transfer the whole 4GB-partition at once (not shown), the system stops serving client operations for 10 minutes.

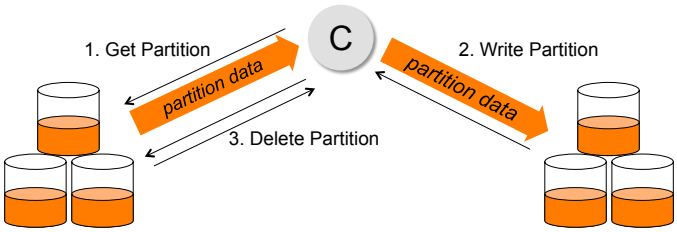
In conclusion, despite its modularity, this protocol is clearly too disruptive and slow to be used in a practical elastic system.

2.1.2 A reconfiguration-based solution

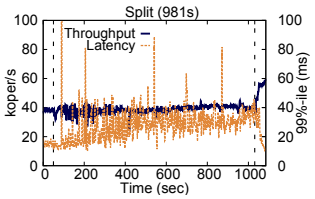
Another solution to dynamically manage partitions with no significant modifications on the replication library is to make use of SMR group reconfiguration, present in many practical protocols [9], [37], [40], [44], [50], [57]. This solution is being implemented for splitting groups in CockroachDB [2], an open-source version of Google Spanner [18] (which is described in §6).

SMR reconfiguration protocols allow the addition, removal and replacement of replicas within a *single group*. Adding a

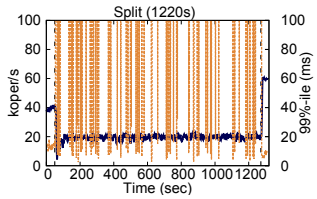
1. Despite its name, BFT-SMaRt can be configured to tolerate only crash faults, using a protocol similar to Paxos [37]. This is the configuration used in this paper.



(a) Client-based partition transfer.

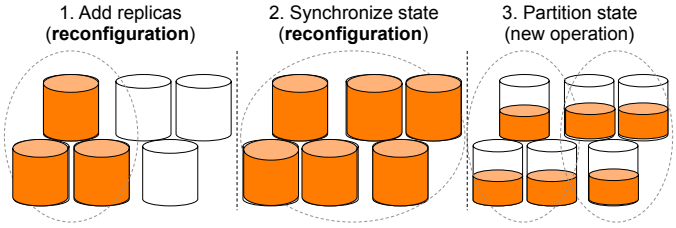


(b) Blocks of 4MB.

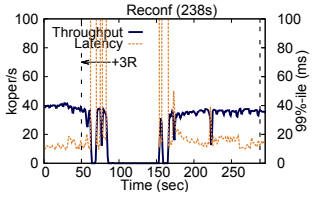


(c) Blocks of 16MB.

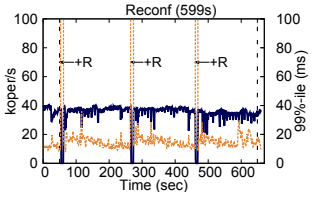
Figure 2. A client-based protocol for partition transfer between RSMs with throughput and operation latency observed by clients during a 4GB-partition transfer operation (in multiple blocks of 4MB and 16MB).



(a) Reconfiguration-based partition transfer.



(b) Adding 3 replicas at once.



(c) Adding 1 replica at a time.

Figure 3. A reconfiguration-based protocol for partition transfer between RSMs with throughput and operation latency observed by clients during a 4GB-partition transfer operation.

replica to a group means that the replica starts participating in the SMR ordering protocol, thus being able to process client requests. However, before processing such requests, new replicas must also be brought up to date by retrieving the state from the other members of the group. In contrast, removing a replica means that it stops participating in the ordering protocol.

Figure 3(a) illustrates the three main steps required for executing a split on a group of three replicas. First, three replicas are added to the source group using SMR reconfiguration. Second, and still within the reconfiguration, new replicas receive a copy of the RSM state. The experiments use BFT-SMaRt's "classical" state transfer protocol [9], [14], in which the new replica fetches the service state from one of the old configuration replicas and validates it using hashes obtained from other f replicas from the same configuration. Apart from the hash validation, this protocol is similar to what is employed in popular protocols and systems like Paxos [37], RAFT [50] and Zookeeper [30]. When the state transfer completes, a message is sent to all replicas to update their state metadata in a way that each half of the state is served by half of the replicas.[2] In the end, each replica changes its replication layer to consider as its group only the replicas responsible for the same part of the state.

To understand the limitations of this protocol, we executed an experiment similar to the one described before, i.e., a 4GB-partition transfer. However, in this case it is important to mention that the replica group hosts an 8GB state, and that this state is transferred in the second step of the protocol (see Figure 3(a)). In this experiment, we were only concerned with the effect on the performance during the first two steps of the protocol, as the third one implies no substantial data transfer (see Figure 3(a)). Figures 3(b) and 3(c) show the effect on the throughput and latency during the reconfiguration of the replica group.

2. In BFT-SMaRt, reconfigurations are initiated by a special client with administrative privileges, as explained in [9]. In the experiments, this client commands the group to add the new replicas and, when the reconfiguration is complete, informs all the replicas about which half of the state it should discard (with another special command).

When three replicas are added in a single reconfiguration (represented by +3R in Figure 3(b)), the complete reconfiguration operation, including adding the new replicas and transferring the entire state to them, takes nearly 4 minutes, and has huge negative effects on the system performance. In particular, the system stops processing requests for more than 60 seconds. This happens because as the source group size suddenly increases to six replicas, the required quorum (simple majority) for ordering requests increases to four replicas, of which one is necessarily new. Given that a newly added replica is only able to process requests after it recovers the group state, the system will stop until the 4GB-state transfer is completed. These side effects of state transfer in RSMs were also studied in previous works [8].

A natural idea to avoid such an undesirable effect is to add replicas one at a time, allowing a newly added replica to complete the reconfiguration procedure before adding another one. More precisely, we start another reconfiguration only after the joining replica finishes installing the checkpoint, replaying the log and is ready for processing requests to group. Figure 3(c) shows an execution of this setup, with each replica addition represented by a +R. The figure shows that, instead of having a long period in which the system stops, there are three short spikes/drops on the system latency/throughput. However, the drawback of this approach is that the full reconfiguration takes 10 minutes to complete (more than three minutes per reconfiguration), 2.4× more than with the previous strategy.

Overall, the key problem of the reconfiguration-based partition transfer is that the new replicas first have to receive the whole state, and only then split the group in two (discarding the unnecessary portion of the state).

2.2 Partition transfer in Non-SMR Databases

Given the impressive amount of work on elastic/cloud-enabled databases (see discussion in §6), an interesting question would be if the techniques employed in these services could be a solution for transferring partitions in RSMs. Systems like Cassandra [36], PNUTS [16] and Dynamo [22] support split and merge operations

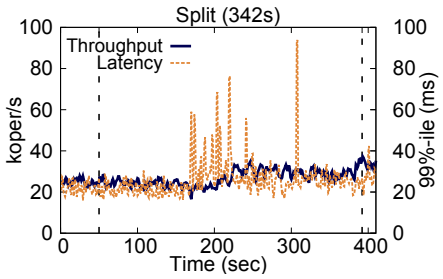


Figure 4. Throughput and operation latency observed by clients during an 8GB state redistribution in Cassandra.

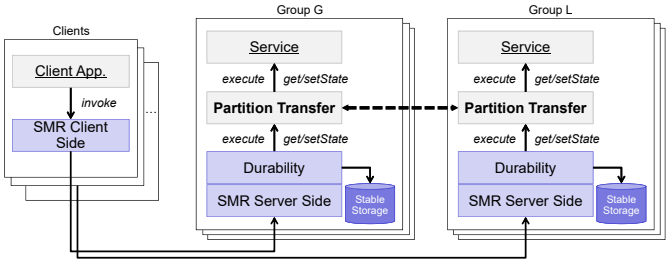


Figure 5. A partitionable and durable replicated state machine.

to handle changing workloads. However, even though the weakly-consistent replication protocols employed in these systems favor a scalable and elastic design, some recent works show that both their latency and throughput are significantly affected during reconfigurations of the replica set [17], [19], [23], [59].

We investigated how Cassandra [36] behaves when a three-replica group hosting an 8GB database (in each replica) is scaled out to six replicas (with 4GB each). The experiment considers the same read-heavy workload used before, with 200 clients accessing the database. This number of clients was enough to make the system reach peak throughput (without substantially increasing the latency) in our setup. After an initial warm up phase, we added three more servers, one at every 2 minutes, respecting the recommendations for the system [1]. To ensure consistency guarantees close to an SMR system, we configured the system to use a replication factor of 3 and quorums of 2 servers for both reads and writes.

Figure 4 shows the 99-percentile of the operation latency and the throughput during this experiment using disks. The results show that the whole process took nearly 6 minutes to complete. This happens because Cassandra employs a conservative design in which data transfers are judiciously done in the background to minimize performance disruptions, which nonetheless occur, as can be seen by the latency spikes.

Other popular elastic databases implement such conservative design, and thus suffer from similar problems [17], [23]. In fact, the idea of performing state transfers in the background has been used in other production systems for implementing replica recovery or reconfigurations (e.g., Zookeeper [30]), following the same rationale.

In conclusion, despite these elastic databases being able to adapt their size to address changes in demand, such conservative design will always lead to a high setup cost for such systems.

3 Partition Transfer for RSMs

A key contribution of this paper is the introduction of a *partition transfer* primitive and protocol in the SMR programming model. This primitive allows a replicated state machine G to transfer part of its state to another replicated state machine L, respecting the following requirements:

1) *Protocol agnosticism*: RSMs require protocols that order requests for implementing coordinated state updates and ensuring strong consistency. These protocols are complex to understand and far from trivial to implement [15]. A requirement of our solution is to not change the SMR protocols and use them as black-boxes for supporting ordered request dissemination in RSMs.
2) *Preserve linearizability*: A fundamental property of a RSMs is that it implements strong consistency. We want to maximize parallelism between operation execution and partition transfer without sacrificing the consistency of the service (i.e., linearizability [29]).
3) *Performance*: In the same way, we want to minimize the performance perturbations on the system during a partition transfer, while we minimize the time required for transferring a partition in modern datacenter setups.

In the following we first describe our assumptions about the environment and the elastic service (§3.1) and then present the partition transfer protocol (§3.2) and its correctness proof (§3.3). We conclude the section with a discussion of how to integrate our protocol with two existing approaches for executing multi-partition operations (§3.4).

3.1 System Model

3.1.1 Environment

We consider a system with an unlimited number of processes, which can be either clients accessing a service or servers implementing the service, that can be subject to Byzantine faults. Therefore, the proposed protocol can tolerate state corruptions and thus be used in BFT systems [9], [14] as long as the total order multicast primitive employed tolerates Byzantine faults. Alternatively, if applied to a crash-tolerant replicated state machine, our protocol only tolerates crashes but still preserving some state corruption validation due to its BFT design.[3]

Additionally, we require the standard assumptions for ensuring liveness of RSM protocols [14], [37], [50]. Processes communicate through *fair channels* that can drop messages for arbitrary periods but, as long as the message keeps being retransmitted, it will be received in the destination [41]. In terms of synchrony, we assume a *partially synchronous* distributed system model [24] in which the system can behave asynchronously for an unknown period of time, not respecting any time bound on communication or processing, but eventually will become synchronous.

Last but not least, we assume the existence of an external trusted component that can trigger the partition transfers on the system, as is the case for other group reconfiguration operations (e.g., join, leave) [9].

RSM Partition Transfer Protocol

(1) *ptransf(PS,L)* is TO-received by group G

(2) Each replica R_i of G sends the state S corresponding to partition spec. PS to its pair R'_i in L, and $H(S)$ to the other members of L

- From now on, every update on state S will be logged in Δ

(3) Replica R'_i in L accepts S when it is fully received together with f matching hashes from other replicas of G. R'_i sets its state to S and TO-multicast an ACK to group G

- If $f+1$ replicas send the same hash h, not matching H(S), R'_i fetches the state S' matching h from some replica from G or L (natural candidates being the replicas that sent h).

(4) When R_i of G TO-receives n-f ACKs from replicas in L, it sends Δ to its pair replica R'_i in L and $H(\Delta)$ to the other replicas of L

- From now on, replicas of G stop serving requests related to PS and redirect such requests to L
- Requests related to PS received by replicas of L are put on hold

(5) A replica R'_i in L accepts Δ only if it receives f matching hashes from other replicas of G. R'_i then applies Δ to its state and sends an ACK to the replicas of G

- If $f+1$ matching hashes (different from $H(\Delta)$) are received, R'_i fetches a matching update list from some replica from G or L
- The replicas of L start processing the PS-related requests that were on hold

(6) When a replica R_i of G receives n-f ACKs from L, it sends an ACK to the invoker of *ptransf(PS,L)*

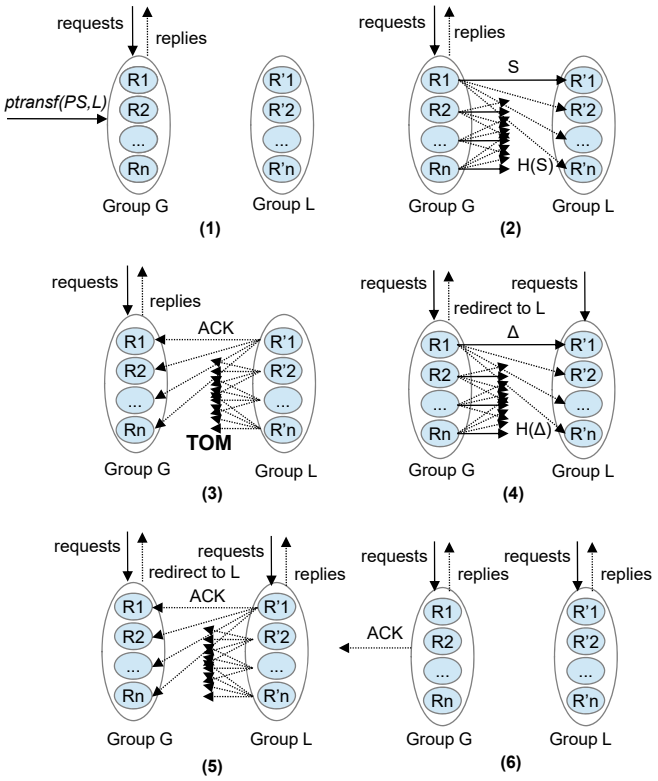


Figure 6. The partition transfer (*ptransf*) protocol.

3.1.2 Service

Figure 5 illustrates the elastic SMR service. Clients invoke operations on a service by sending messages to the set of servers hosting it. We assume the existence of some application-specific mechanism to allow clients to find the appropriate servers to which direct the request. Servers are organized in groups providing a stateful service as a replicated state machine. Each group of servers has n replicas and tolerates up to f faults. We refer to group G as replicated state machine G. A RSM is accessed through a replica coordination protocol that provides strong consistency. In practice, this means a consensus protocol will be used to ensure that requests are delivered to the replicas in *total order (TO)* [55]. We use *TO-multicast* and *TO-receive* to denote the transmission and reception, respectively, of messages in total order. *TO-multicast*, in particular, can be used by replicas in a group to send specific partition transfer protocol messages to the replicas in another group.

We assume that the service state S_G of group G can be partitioned in a number of *closed partitions* $s_1, ..., s_p$, in the sense that operations (and in particular, updates) submitted to the service affect only one of these partitions. For example, a key-value store can be partitioned in several key ranges such that each *put*/*get* will be executed on only one of them.

In the next section we describe the partition transfer protocol considering only operations performed on a single partition. Later on we explain how the protocol can be slightly modified to be integrated with multi-partition operations, thus relaxing the assumption on operations in closed partitions.

3. In the evaluation of the protocol we considered its application to replicated systems tolerating only crash failures, since this allows reaching higher levels of system throughput and hence creates more stressful scenarios for the purpose of evaluating the reconfiguration performance.

3.2 Partition Transfer Protocol

The partition transfer protocol is encapsulated in a primitive $ptransf(PS, L)$ that can be invoked in a replicated state machine G to make its replicas transfer the partition specified in PS (e.g., a key-range in a key-value store [27], a directory in a file system [7], a subtree of a hierarchical data structure [30], a set of tables in a distributed database [18]) to a replicated state machine L.

Figure 6 presents and illustrates the six-step protocol for implementing *ptransf*. As mentioned before, we consider that *ptransf* requests (Step 1) are issued by an application-specific and trusted external component, in the same way as other reconfiguration requests [9]. When a group receives a *ptransf* request, each replica verifies whether the request was issued by such authorized component and only initiates the partition transfer protocol after this validation. The core idea of the protocol is to leverage the fact that all correct replicas (of both groups involved in the partition transfer) execute their operations in total order, mimicking a centralized server. In this way, it is possible to ensure that all correct replicas of G execute operations in PS until a certain point in the totally ordered history of executed operations, and then start redirecting further requests for this partition to the new group L (Step 4). Since partitions can be arbitrarily large, we use copy-on-write to make G store all executed updates in the partition specified in PS during its transfer to L (Step 2). Such updates are transferred after the state is received by a quorum of replicas in L (Step 5) and *ptransf* concludes (Step 6). The objective is to minimize interruptions to request serving in PS – these will occur

only during the transfer of the updates Δ that were executed during the state transfer. Note that the size of Δ is expected to be much smaller than the partition size itself, which means that both the time taken to complete Step 4 and the extra memory space needed to store Δ are typically small.

Several considerations can be made about the partition transfer protocol. First, our solution is completely modular with respect to the SMR protocol [37], [14], [9] or even the durability strategy [8] implemented in the system. This is quite important for reusing the already available protocols.

Second, we opted to perform full partition transfers between pairs of replicas, having one single replica in the source group transmit to one single replica in the destination group. This design option makes our system bandwidth-efficient in multi-rack and virtualized environments. This pairwise transfer is executed after replicas of G connect with their corresponding paired replica in L (Step 2), which is done through a deterministic bijective function mapping every replica from G to one replica from L. We envision scenarios in which the replicas of a group will be deployed in different racks (or physical machines) to avoid correlated faults and will have to transfer a partition to another set of replicas deployed in the same racks (or physical machines), ensuring such transfer will be done within the rack network boundaries, without using the (usually) oversubscribed network core.

Third, even if only crash faults are considered, we opted to provide a more general Byzantine-resilient protocol in which data corruptions are detected and recovered (through hash comparisons and state fetching, in Steps 3 and 5). This is important to ensure that even under the most uncommon failure modes [28] the destination group will still start accepting operations for the transferred partition with all its correct replicas in the same state. More specifically, at the end of the protocol at least $n-f$ replicas of L have the correct state for the partition. Note that the protocol does not preclude the case in which some correct replica of group L finishes the protocol with an invalid state. However, there will be a sufficient number of correct replicas in L with the correct state (independently of the pairings with replicas, as faulty transfers are recovered by correct replicas of G – see sub-bullet of Step 3), and thus normal state transfer protocols for durable state machine replication can be applied, both in the crash [37], [50] and Byzantine fault models [8], [14]. This implies that all replicas in the group can start processing requests in the same state [55].

Fourth, consensus is employed only in two steps of the protocol (but encapsulated in the total order request delivery of the RSM): Step 1, when the *ptransf* primitive is invoked, and Step 3, when all replicas of L invoke RSM G to inform about the hash of the received state. Regarding Step 3, when the replicas of G receive $n-f$ matching ACKs from different replicas in total order, they send the same set of updates to the replicas in L. Notice that the $(n-f)$-th matching ACKs will be received in the same point of the execution history of all correct replicas of G, ensuring they will stop executing requests for *PS* in a coordinated way.

3.3 Correctness Argument

The *ptransf* operation must satisfy the following properties:

- **Safety 1**: When *ptransf* completes, the transferred partition will not be part of the source group state and will be part of the destination group state;
- **Safety 2**: Linearizability of the service is preserved by the partition transfer;
- **Liveness**: *ptransf* eventually completes.

Safety 1 is satisfied due to the fact that both the state (Step 2) and the updates (Step 4) are transferred. Moreover, after sending the updates (Step 4), the original group will not execute any other operation for the partition.

Safety 2 is a bit more tricky to show, since we need to take into consideration the definition of linearizability [29]. A service execution is linearizable if every execution history (containing requests and replies for client operations) is linearizable. We say that *an operation is linearizable* if the extension of a linearizable history with its request and reply still makes the history linearizable.

Linearizability is ensured in RSMs due to the fact that all requests are executed in total order by correct replicas. Consequently, every executed operation needs to produce a reply that considers all previously executed operations.

Let G be an initial group of replicas with state $S_{part} \cup S_{rem}$ and L a group of replicas that will receive a partition S_{part}. Recall that the system is partitionable, i.e., every operation either accesses S_{part} (the partition to be transferred) or S_{rem} (the remaining partition), as defined in §3.1. This means that the system is linearizable as long as the sub-histories containing operations for S_{part} and S_{rem} are linearizable [29].

Since the service provided by the RSM G ensures linearizability, the operations for S_{rem} are linearizable. Due to the same reason, the operations for S_{part} executed before $ptransf(part, \mathsf{L})$ are linearizable. When $ptransf(part, \mathsf{L})$ is received, the state S_{part} resulting from all previously executed operation for the partition is transferred to L (Step 2). Every operation for *part* executed between Steps 2 and 3 is executed in G, and linearizability is maintained. These operations are stored in Δ and transferred to L in Step 4. After this point, G stops executing operations for *part*. As specified in Step 5, these operations are only executed by L after this step, when this group already received S_{part} and applied the updates in Δ. Consequently, after this point, these operations will reflect all previously executed operations for *part*, ensuring the linearizability of the partition.

Liveness is satisfied due to the fact that all six steps of the protocol terminate. This happens because (1) the total order multicast primitive employed in the protocol terminates in our system model (Steps 1 and 3); (2) the fair link assumption implies that both the state (Step 2) and the update list (Step 4) can be transferred in finite time; (3) if a replica does not receive the state or update list that matches f hashes, there will be $f+1$ hashes matching some other state and list, and this state can be fetched (Steps 3 and 5); and (4) all correct replicas will send ACKs to the original replicas, and thus these replicas will receive $n-f$ messages to make progress (Step 6).

3.4 Multi-partition Operations

In the following we describe how *ptransf* can be integrated with two existing approaches for supporting multi-partition operations, namely, the S-SMR protocol [10] and Spanner multi-partition transactions [18].

3.4.1 S-SMR

S-SMR [10] considers a linearizable service with its state statically partitioned among several groups (partitions). Each group is composed of a set of replicas that implement a replicated state machine. Single-partition operations are executed only on

individual groups, but the system also supports multi-partition operations, which require coordination of multiple groups.

More specifically, a multi-partition operation works as follows. First, the client determines which groups must participate in the operation and sends the operation to each of these groups using *TO-multicast*. Second, after ordering the requests (individually, on each group) the groups exchange all the necessary data to execute the operation in both groups. Next, each group executes the operation and signals its completion to the other groups. After all groups involved in the operation were signalled, the result is returned to the client. The basic trick here is to block conflicting operations and execute them in sequence, preserving linearizability [10].

To integrate *ptransf* with S-SMR, we need to make two modifications on the original protocol. First, after a reconfiguration, the system needs to update the partition metadata to allow clients to find the new partitions. Second, multi-partition operations cannot be processed while Steps 4 and 5 of *ptransf* are being performed in the source group G. Operation execution can only be resumed after this group receives $n-f$ totally-ordered ACKs from the destination group L. This guarantees that the partition state and its differences were received by L, ensuring the service consistency.

When the partition transfer finishes, source group G may still hold multi-partition operations that cannot be executed since it is no longer responsible for the data required by them. In this case, G signals the other groups involved in the operation, and the client receives a redirect message, restarting the multi-partition operation in the updated groups.

3.4.2 Spanner

Similarly to S-SMR, in Spanner [18] the application state is partitioned in tablets hosted on several groups. Each group is composed of a set of replicas that implement Paxos state machines to maintain shards of a database.

The steps to process a multi-partition transaction are the following. First, a client communicates with a proxy location to determine which groups maintain the data touched by the transaction. Second, the client retrieves this data from the groups, acquiring locks for them. Next, the client executes its transaction locally, chooses one of the groups involved in the transaction as a coordinator group C, and sends the result and the id of C to all groups involved in the transaction. Finally, group C coordinates a two-phase commit with the other groups for committing the transaction.

To integrate Spanner transactions with *ptransf* we need to slightly modify Steps 4 and 5 of our protocol to account for the way locks are managed in Spanner. More specifically, while the partition is being transferred, its associated data must be locked. The source group only releases the locks after receiving $n-f$ totally-ordered ACKs from the destination group. This guarantees the consistency of the service as the partition state and its differences are received by the other group. When a client tries to access the data in the transferred partition, it receives a redirect message and aborts the multi-partition transaction. Eventually, the proxy location server is updated and the client can re-execute the transaction in the updated groups.

4 Implementation

We implemented our partition transfer protocol in an elastic storage infrastructure called CREST, whose architecture is depicted in Figure 7.

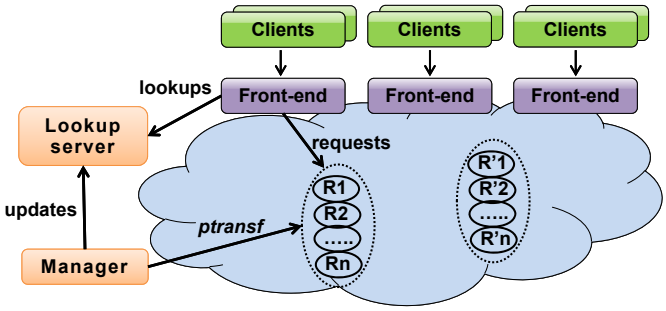


Figure 7. CREST architecture.

In CREST, clients send requests to stateless *Front-ends* exposing a key-value interface including operations such as *GET* and *PUT*. The Front-ends use the *Lookup server* to discover which group serves a certain key. All data is sharded across the groups. The *Manager* triggers reconfigurations by sending *ptransf* commands to the groups.

We implemented the *ptransf* protocol on top of the BFT-SMaRt [9] replication library. Our implementation follows the model described in Figure 5, with the partition transfer layer implemented between the durability management [8] and the service (in this case, a key-value store). The pairwise transfer of the partition state and update list (see Steps 2 and 4 in Figure 6) is implemented over dedicated sockets, to minimize interference with the normal processing of client operations. To transfer the key-value store partitions we used a serialization library called Kryo [4]. All updates and partition transfers are written to the replicas log, in durable storage. We parallelize the writing of state to the storage device (including buffer flushing) with other steps of the protocol, leveraging the fact that (1) the original group will only delete the transferred state after Step 5, and (2) the receiving group disk is idle during a state split since it only starts executing and logging requests after Step 5.

The Lookup server and the Front-ends were implemented as Java servers, and the Manager as a Java client. As already mentioned, a Front-end is just a soft-state proxy. The Lookup server holds a map between intervals of keys and groups. For each Front-end request, the Lookup server returns a key interval and a group. In this way, the number of lookup operations can be minimized as the Front-end caches key locations. The Manager is responsible to send *ptransf* commands to the groups in order to perform reconfigurations. After every reconfiguration process, the Manager updates the Lookup server map.

Our current implementation of the Manager and Lookup server are not fault-tolerant, as centralized servers were sufficient for evaluating the *ptransf* protocol. However, this is not a fundamental limitation. The Lookup server can be made fault-tolerant by implementing it as a key-value store on top of an SMR library. The Manager, on the other hand, requires a bit more work as it interacts with the other components as a client and not as a server. Therefore, a fault-tolerant solution requires a set of Managers, one being active and the others serving as backups, keeping the service state replicated using a RSM protocol. Notice that implementing a BFT manager will require a more complex approach, which is out of the scope of this paper.

5 EVALUATION

We evaluate our partition transfer algorithm within CREST by assessing the duration of a partition transfer and how it affects the service throughput and operation latency observed by clients generating different workloads on the system.

5.1 Setup

All experiments were conducted in a cluster of 18 machines interconnected by a Gigabit Ethernet switch. Each machine has two quad-core 2.27 GHz Intel Xeon E5520 CPUs with 32 GB of RAM, a 146GB 15k-RPM SCSI disk and a 120GB SATA Flash SSD. The machines run Linux Ubuntu Trusty with kernel version 3.13.0-32-generic for x86_64 architectures with Oracle Java 1.7.0_80-b15. In these experiments we avoided using VMs and dynamic resource configuration managers to capture the performance of the partition transfer protocol without the VM setup overhead.

Unless stated otherwise, the experiments were executed with BFT-SMaRt configured for *crash fault tolerance*, with groups of three replicas (tolerating a single fault).

5.2 Methodology

The experiments were conducted with a database with 8GB where small-string keys were associated with 4kB values. We used two YCSB workloads [17]: the *read-heavy workload*, with 95% of *get* and 5% of *put* operations (95/5), since it is similar to what is reported in several production systems [12], [16], [18] and in some related works (e.g., [59]); and the *write-heavy workload*, with an equal distribution of puts and gets (50/50) to exercise our protocol under a heavy update load. Clients access keys following a Zipfian distribution. This setup and workload creates a reasonable but demanding scenario for an SMR-based storage system [9], [8], [11].

Our experiments consider partition transfers of 4GB (half of the state in the key-value store), either in a single *ptransf* execution or in multiple executions of *ptransf*, each transferring blocks of 4MB, 16MB, and 256MB (approximately 1024, 256, and 16 protocol executions, respectively).

5.3 Partition Transfer on an Idle System

Our first experiment measures the time our protocol takes to transfer 4GB of state between two RSMs without any client-imposed workload. The results are presented in Table 1, which also contains the duration of the same transfer using the client- and reconfiguration-based solutions described in §2.1, considering the state maintained in both SSDs and disks. We also present, as a reference, the duration of a 4GB-file transfer between two machines using *rsync* [5], a widely-used tool for synchronizing files between machines. Although the size of the partition was the same in all cases (4GB), we presented results for different block sizes for each solution.

The results show that *ptransf* is 8×-16× and 2×-6× faster than the client- and the reconfiguration-based solutions, respectively. Our protocol is much more efficient than the client-based protocol as it does not have a single process (the client) as a bottleneck. Furthermore, it is also significantly more efficient than the reconfiguration-based protocol due to two factors: (1) the transference is done in a pairwise way, and (2) only the data specified by the partition (4GB) is transferred, not the whole state (8GB). On the other hand, *ptransf* is 20% and 23% slower (for disks and SSDs, respectively) than a two-machine synchronization using rsync.

System	*Disks*	*SSDs*
client (4MB)	802.3 ± 3	823.3 ± 1.1
client (16MB)	855.0 ± 3.78	873.0 ± 5.0
reconfig (+3R)	201.0 ± 8.1	209.6 ± 11.5
reconfig (+R+R+R)	294.0 ± 4	307.3 ± 2.3
ptransf (4MB)	100.6 ± 0.89	114.4 ± 1.34
ptransf (16MB)	85.8 ± 0.84	97.6 ± 2.07
ptransf (4GB)	54.0 ± 1	64.4 ± 2.97
rsync	44.8 ± 1.30	52.2 ± 0.84

Table 1
Duration of a 4GB partition transfer (in seconds) using *ptransf* and alternative solutions (see §2.1) in an idle system.

Notice that the reported performance using disks was better than using SSDs. This happens because our disks have a better throughput than our SSDs for sequential writes: 130 vs 120 MB/s.

5.4 Partition Transfer on a Saturated System

The next set of experiments aims to shed light on the impact that a partition transfer can have on the performance of a saturated system. The objective is to understand how triggering a partition transfer (e.g., for scaling-out) under critical conditions affects the latency and throughput of the system.

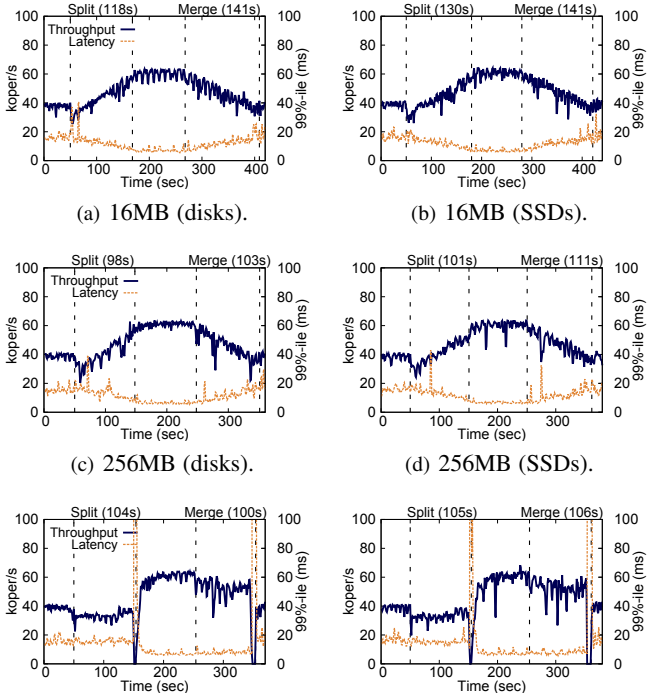
In order to define the conditions for system saturation, we progressively launch a number of clients until the system reaches its peak throughput for a single replica group. Adding more clients after this point only increases the latency. We identified that the system achieves the peak throughput for read-heavy and write-heavy workloads with 140 (≈40000 4kB-oper/s) and 70 (≈9000 4kB-oper/s) clients, respectively. The observed peak throughputs are similar with disks and SSDs.

In all experiments, when the system reaches its peak throughput we start a partition transfer to split the state of the system to another group. One hundred seconds after the split completes, we invoke another partition transfer to merge the state of the second group back with the first.

5.4.1 Read-heavy workload

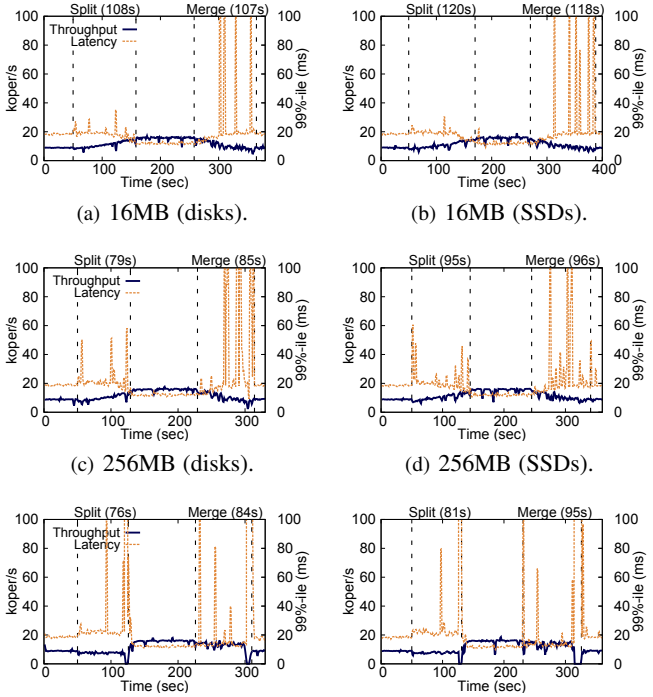
Figure 8 shows the throughput and the 99-percentile operation latency (obtained from 2-second intervals) observed by clients running the 95/5 workload using disks and SSDs.

Three aspects are worthy of consideration from these executions. First, the state split tends to be faster than the state merge (the second transfer) using both disks and SSDs. This happens because the split transfers the partition to an idle group (not yet receiving client operations), while the merge transfers the state to a busy group. Second, as in idle setups, the partition transfers are faster with disks. Third, there are fewer spikes in the throughput and in the operation latency when the partition is transferred in small blocks. For instance, with a 16MB block size, the throughput starts to increase a few seconds after the split initiates, as clients start sending commands to the second group, decreasing the load on the first. In contrast, with a 4GB block size (a single *ptransf* execution), the throughput starts to increase only a few seconds after the split completes, while the operation latency increases significantly during the partition transfer. This happens because the group is fully loaded with client operations



(a) 16MB (disks). (b) 16MB (SSDs).
(c) 256MB (disks). (d) 256MB (SSDs).
(e) 4GB (disks). (f) 4GB (SSDs).

Figure 8. CREST throughput and operation latency in saturated conditions with reconfigurations using disks and SSDs. Read-heavy (95/5) workload.



(a) 16MB (disks). (b) 16MB (SSDs).
(c) 256MB (disks). (d) 256MB (SSDs).
(e) 4GB (disks). (f) 4GB (SSDs).

Figure 9. CREST throughput and operation latency in saturated conditions with reconfigurations using disks and SSDs. Write-heavy (50/50) workload.

while transferring the whole partition and keeping track of the updates being executed.

5.4.2 Write-heavy workload

Figure 9 shows similar executions but now considering the 50/50 workload.

The behavior observed in these experiments is similar to that discussed for the read-heavy workload, with two noticeable differences. First, the latency is slightly higher and the throughput is substantially lower than in a read-heavy workload. Second, there are more latency spikes during splits and merges, both for disks and SSDs. This happens because now there are more updates, which imply more operations being written to the write-ahead log of the replicas [8] and increased I/O contention with the partition write at the receiving group. These spikes tend to be more common during merges than during splits. The explanation comes from the different nature of splits and merges. During a split, the state is transferred to an initially empty RSM, which is progressively being loaded, as blocks of the partition are transferred. During a merge, on the other hand, the state is transferred to a group already loaded, which causes higher I/O contention. This should not be a problem in practice since splits are executed under high loads, while merges are triggered for consolidating resources, when the system is mostly idle.

Notice that when using 4GB blocks (Figures 9(e) and 9(f)), the throughput of the system goes to zero by the end of the *ptransf* operations. This happens because in this write-heavy experiment with a single big-block transfer, the size of Δ transferred in Step 4 of the protocol is around 300MB, blocking the system for almost 3 seconds. This size of Δ also represents the amount of additional memory needed for using copy-on-write on this experiment. It is worth to remark that this is a worst-case scenario – with smaller block sizes there are several significantly smaller Δ transfers and no noticeable blocking in the system.

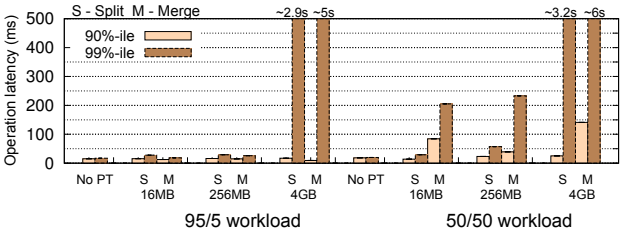
5.4.3 Consolidated results

We now show consolidated results for 10 executions of splits and merges using each configuration, comprising almost 4 hours of partition transfers. As in previous experiments, these results (presented in Figure 10) consider different workloads and block sizes. However, from here on we only show results for setups using disks as the results using SSDs lead to the same insights.

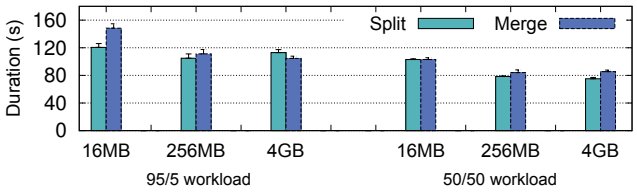
Figure 10(a) reports the 90th and 99th latency percentiles for periods with and without partition transfers (noted in the figures as "Split" and "Merge" operations for different block sizes and "No PT", respectively) for 95/5 and 50/50 workloads. The 90-percentile latency represents the behavior that most clients observe during a split or a merge, while the 99-percentile captures the effect of spikes on the service level metric. Notice that the 99-percentile latency values for 4GB blocks are very high and hence we represent them numerically, over the corresponding bars (both in Figure 10(a) and Figure 11(a)).

The results confirm the trends observed in the previous section. In particular, splits and merges have no significant effect (when compared to the "No PT" situation) on the 90-percentile latencies, under the 95/5 workload. This is not true for write-heavy workloads, as the latency of merges are consistently higher (due to I/O contention).

As expected, the 99-percentile latencies are affected during splits and merges for all block sizes and workloads. However, the values during splits are still under 60 ms, which is way below real-world SLAs defining the 99-percentile under 100 ms [16],

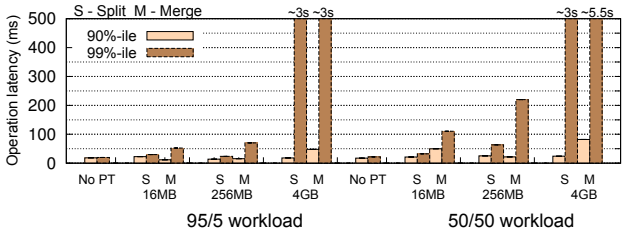


(a) Operation latency during partition transfers (groups of three replicas).

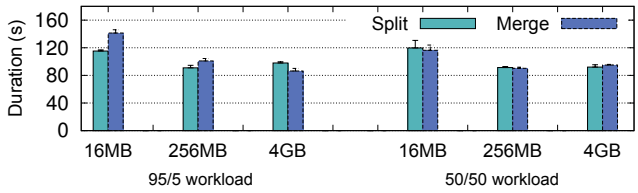


(b) Duration of partition transfers (groups of three replicas).

Figure 10. Latency during 4GB-partition transfers and the duration of such transfers using disks and groups of three replicas ($f = 1$) and considering different block sizes and workloads.



(a) Operation latency during partition transfers (groups of five replicas).



(b) Duration of partition transfers (groups of five replicas).

Figure 11. Latency during 4GB-partition transfers and the duration of such transfers using disks and groups of five replicas ($f = 2$) and considering different block sizes and workloads.

[22]. The effect is much more dramatic for merges, for the reasons discussed before. However, such effects have few practical implications as merges are usually executed in idle systems (as also discussed before).

The only exception to the general trends discussed before is observed when using 4GB blocks. In this case, the 99%-ile latency increases 100× or even more, as clients block during the transference of the Δ in Step 4 of *ptransf*. This is illustrated in Figures 8(e) (gaps at seconds 150 and 350) and 9(e) (gaps at seconds 120 and 290).

Figure 10(b) presents the average duration of split and merge operations (together with standard deviation) during the previous experiments. The results show that when considering a specific configuration, the amount of time required for executing the partition transfer is stable across different executions, as attested by the negligible standard deviation. Additionally, the results show that transferring 16MB blocks lead to higher partition transfer durations than when using other block sizes in all configurations. This happens mostly because the *ptransf* protocol runs more times, leading to more protocol messages being exchanged and more synchronization points (Steps 4 and 6 in Figure 6). However, the duration of splits and merges are similar for blocks of 256MB or 4GB (whole partition in a single transfer). This indicates that increasing the block size after a certain value does not lead to faster partition transfers, instead, it only makes the latency worse (as shown in Figure 10(a)).

5.4.4 Comparison with alternative solutions

Table 2 compares the duration of a split using the best configuration of *ptransf* (256MB blocks) and the alternative solutions discussed in §2.1 for a saturated system. By comparing this table with the Disks column in Table 1, it is possible to see that all protocols have a noticeable increase on the duration of split.

The results show that a split using *ptransf* is 9.5× and 11.7× faster than the client-based solution (Figure 2), and 2.3× and 5.4× faster than the reconfiguration-based solution (see Figure 3) for a read-heavy workload. The difference is even bigger with a write-heavy workload.

System	*read-heavy*	*write-heavy*
client (4MB)	999.0 ± 15.9	871.3 ± 11.7
client (16MB)	1236.0 ± 16	1198.0 ± 9
reconfig (+3R)	243.5 ± 6.6	270.4 ± 2.3
reconfig (+R+R+R)	575.0 ± 21	623.0 ± 10.4
ptransf	104.8 ± 6.3	78.4 ± 0.9

Table 2
Duration of a 4GB partition transfer (in seconds) using *ptransf* and alternative solutions (see §2.1) in a saturated system using disks for read- and write-heavy workloads.

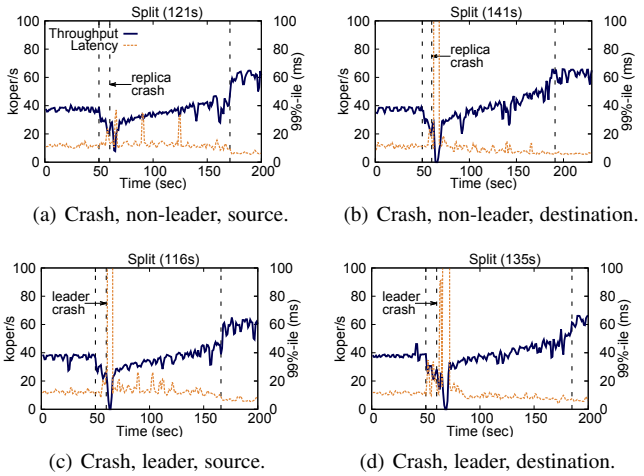
Even more importantly than these results is the fact that our protocol causes less perturbations on the latency and throughput observed by clients, as can be seen by comparing the executions of Figures 2 and 3 with splits presented in Figures 8 and 9. The only exception is in Figure 3(c), for the three single-replica reconfigurations, in which the effects are negligible, but the transfer latency is 5× higher than with our protocol.

5.5 Partition Transfer in Bigger Groups

The results presented up to this point consider groups with three replicas, that tolerate a single server failure ($f = 1$). In this section we present similar experiments, but now considering groups of five replicas ($f = 2$), to understand the behaviour of our protocol with bigger replica groups. In this setup, we identified the system achieves the peak throughput for read-heavy and write-heavy workloads when 120 (≈37 koper/s) and 50 (≈7 koper/s) clients are used, respectively. Figure 11 presents the results.

When compared to the results for groups of 3 replicas (Figure 10), the results in Figure 11 show exactly the same trends, despite some small variations in the latency.

Regarding the duration of the partition transfers, the results are also similar to the three replica setups. This happens because our protocol transfers state of replicas in parallel, pairing each replica



(a) Crash, non-leader, source. (b) Crash, non-leader, destination.

(c) Crash, leader, source. (d) Crash, leader, destination.

Figure 12. Four failure scenarios during a split using a 256MB block size and disks.

on the source group with another replica on the receiving group. This design makes the data transfer phases of the protocol (Steps 2 and 4 in Figure 6) – which dominate the duration of a transfer – mostly independent from the group size.

5.6 Faults during the Partition Transfer

In this section we discuss how replica failure scenarios affect the *ptransf* protocol. We consider four failure scenarios: (a) a non-leader replica of the source group crashes, (b) a non-leader replica of the destination group crashes, (c) the leader replica of the source group crashes, and (d) the leader replica of the destination group crashes.

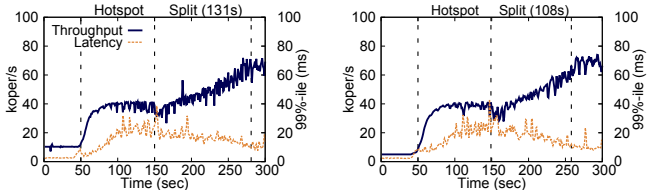
In this experiment, we considered a group of three replicas, a read-heavy (95/5) workload and blocks of 256MB. The experiment focuses on the split operation since it is normally executed when the system is under stress and needs to scale out as fast as possible. In all failure scenarios we crash the target replica 10 seconds after the split begins and use a timeout of 2 seconds for suspecting a leader and starting the leader election protocol. Figure 12 presents representative executions for the four scenarios.

When comparing the executions with a non-faulty execution (left half of Figure 8(c)), it is clear that failures have a noticeable effect on the partition transfer.

Comparing the scenarios where a non-leader replica crashes (Figure 12(a) and 12(b)), the negative effects on the system performance are more prominent when the replica belongs to the destination group. This happens due to a combination of two factors. First, during a split the operations targeting the partition being transferred are progressively being redirected to the destination group, with each block transferred. Second, a high percentage of the reads performed by clients on the two surviving replicas of the destination group are not completed using the consensus-free optimized read, and need to be retried using the total-order protocol [9], [14].

Crashing the leader replica of both source and destination groups (Figures 12(c) and 12(d)) leads to the same effect in the system: the group with the faulty replica stops processing requests for 2 seconds (the timeout value), until a new leader is elected and normal processing is resumed.

In summary, the results show that the performance of the partition transfer protocol is robust against failures, and the most visible negative effects are due to performance degradation caused to the ordering protocol of the RSMs.



(a) 16MB block size. (b) 256MB block size.

Figure 13. Scale-out in a hotspot group using disks.

5.7 Partition Transfer in a Hotspot

In previous experiments we performed splits and merges on saturated RSMs to evaluate how our protocol works in a system on its limits. In this final experiment we discuss the effects of doing a partition transfer on a hotspot group, i.e., a group of replicas that experiences a sudden and large increase in its workload. We create such hotspot by uniformly increasing the number of clients, from 20 to 200, over a hundred seconds interval, which also results in a tenfold latency increase (from 2.5 ms to 25 ms). This hotspot scenario is similar to the one used in [59], and was inspired on the statistics of the "9/11 spike" experienced by CNN.com [39], where the workload increased by an order of magnitude in 15 minutes. For this experiment, we consider groups of three replicas and a read-heavy workload.

Figure 13 shows the throughput and the latency when a split is performed on the hotspot group using two block sizes. As can be observed, the split causes a similar effect on the throughput and the latency as when the group is saturated (Figure 8(c) and Figure 8(e)). Using 16MB blocks to transfer the partition makes the split 20% slower than using 256MB blocks, but the overall effect on latency and throughput is the same.

In conclusion, the results show that our *ptransf* implementation allows a 8GB storage system to double its capacity when subject to an unusual high demand without any significant performance disruption, taking less than two minutes to scale out.

6 Related Work

6.1 SMR Scalability

Several SMR protocols support the addition and removal of replicas at runtime [9], [37], [40], [44], [50], [57]. Other works exploit virtualization technology for starting or replacing replicas very efficiently by employing techniques such as copy-on-write to bring the new replica up to date with minor disruptions on the service [53], [61]. However, these reconfigurations only change the set of replicas in a *single group*, and do not improve the performance of the system since the protocols used for ordering requests are inherently non-scalable. To the best of our knowledge, we are the first to propose a well-defined primitive and protocol for sharding RSMs at runtime.

Different techniques have been proposed to improve the scalability of SMR-based services. Some works try to remove bottlenecks both from the ordering protocol [42], [44], its implementation [54], [6] and the execution of requests [32], [34], [43]. For example, protocols like Mencius [42] and Egalitarian

Paxos [44] try to spread the additional load of the primary replica among all the system replicas. In a complementary way, Santos and Schiper [54] and Behl et al. [6] show that it is possible to substantially improve the performance of an SMR implementation by architecting these systems using multiple threads. Similarly, several works try to parallelize the execution of requests for taking advantage of multi-core servers and improve the RSM performance. The techniques range from identifying independent RSM commands that can run in parallel without endangering determinism [34], [43], to executing the requests (using multiple threads) before ordering them [32]. Ultimately, these approaches help addressing the request ordering and execution bottlenecks (and thus can be combined with CREST to improve the performance of a single group), but the fundamental limitation of every replica executing every operation still remains.

A third group of works aims to scale SMR-based services by dividing their state among several partitions, implemented by (mostly) independent RSMs [10], [18], [27], [38], [51]. Bezerra et al. [10] propose a technique for executing atomic operations spanning multiple partitions still ensuring linearizability. This work was later extended for optimizing data placement in the partition for decreasing the amount of multi-partition operations [38]. Padilha and Pedone propose Augustus [51], a storage architecture that follows a partition approach and tolerates Byzantine faults. This architecture enables transactions across partitions. Although the previously mentioned works ([10], [38], [51]) use partitioning to address the scalability of SMR-based services, there is no support for creating such partitions dynamically.

6.2 Elasticity in SMR-based Systems

Scatter [27] is a consistent distributed key-value store where key ranges are served by groups of replicas. Split and merge reconfigurations are available, but the paper does not mention how the state transfer between partitions is realized, neither measures the impact of such data transfers (the reported values suggest that only a trivially-small state is used in the experiments). Additionally, the Scatter partitioning algorithm works only for adjacent groups in its Chord-like architecture, requiring substantial modifications to the Paxos protocol to implement a multi-group 2PC commit. Our solution, on the other hand, aims for fast and predictable elasticity with multi-gigabyte partitions, targeting thus a general limitation of SMR systems. Moreover, it can be implemented on top of any SMR protocol.

Spanner [18] is a globally-distributed database that shards data across many sets of Paxos-based state machines in Google datacenters. Although not detailed in the paper, Spanner allows shards to be transferred in order to balance load or in response to failures. During these transfers, clients can still execute transactions (including updates) on the database. Similarly to most elastic database systems [16], [22], [36], Spanner transfers partition data slowly to minimize the impact of such reconfigurations on the system performance. When the remaining data to be transferred is sufficiently small, the system uses a multi-partition transaction to move the metadata of the shard between the two groups. Although the protocol appears to protect the safety of the database, the liveness is not guaranteed since moving data in Spanner may take an unbounded amount of time if the update rate is higher than the transfer rate. In this paper we proposed a specialized abstraction and a (safe and live) protocol for transferring partitions as efficiently as possible, and explained how it could be integrated with the multi-partition transactions of Spanner.

6.3 Elastic Databases

There is a large set of works from the database community for implementing elasticity in existing systems [20], [21], [25], [56], [58]. These works propose solutions to split and merge state of sharded databases without violating the ACID properties, and can be broadly divided in terms of the database architecture they consider. Some systems employ shared storage architectures [20], [21], where the database nodes persist data on some shared "always available" infrastructure. In such architectures, when a server is added there is no need to perform a partition transfer, being sufficient to copy only the database cache and active transactions. Other works focus on shared nothing architectures [25], [56], [58], where each partition is kept on different nodes. In this case, reconfigurations require a partition transfer. However, existing works only consider transferences between two servers, not between two groups of servers, as is required for RSMs.

7 Conclusion

This paper introduced a partition transfer protocol for implementing elasticity in replicated state machines. Our protocol minimizes (1) the time required to transfer the partition between two replica groups and, (2) the negative effects of this data transfer on the operation latency observed by clients. The proposed protocol can be integrated in any SMR consensus algorithm (e.g., Paxos [37], PBFT [14], RAFT [50]), since it operates on top of the ordering protocols. Furthermore, it allows the dynamic creation of partitions in different replica groups, complementing some recent works on scalable state machine replication [10], [27], [51].

The proposed protocol was implemented and evaluated in a key-value store. In our experiments, a prototype system using our protocol was able to double its capacity with minimal performance degradation, showing that it is possible to have elastic reconfigurations even for non-trivial partition sizes (e.g., 4GB).

Acknowledgements

The authors would like to thank João Sousa for the support in BFT-SMaRt and the IEEE TPDS reviewers for the excellent comments that helped improving this paper.

References

[1] Cassandra documentation. http://www.datastax.com/documentation/cassandra/2.0/cassandra/gettingStartedCassandraIntro.html.

[2] CockroachDB Design Document. https://github.com/cockroachdb/cockroach/blob/master/docs/design.md.

[3] HydraBase – The evolution of HBaseFacebook. https://code.facebook.com/posts/321111638043166/hydrabase-the-evolution-of-hbase-facebook/.

[4] Kryo - Java serialization and cloning: fast, efficient, automatic. https://github.com/EsotericSoftware/kryo.

[5] The rsync algorithm. http://rsync.samba.org/tech_report/tech_report.html.

[6] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Consensus-oriented parallelization: How to earn your first million. In *Proceedings of the 16th ACM/IFIP/USENIX Middleware Conference – Middleware'15*, 2015.

[7] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. Scfs: A shared cloud-backed file system. In *Proc. of the USENIX Annual Technical Conference – ATC' 2014*, 2014.

[8] Alysson Bessani, Marcel Santos, Joao Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *Proc. of the USENIX Annual Technical Conference – ATC'13*, 2013.

[9] Alysson Bessani, Joao Sousa, and Eduardo Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proc. of the IEEE/IFIP International Conference on Dependable Systems and Networks – DSN 2014*, June 2014.

[10] C. E. Bezerra, F. Pedone, and R. van Renesse. Scalable state-machine replication. In *Proc. of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks – DSN'14*, 2014.

[11] 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*, April 2011.

[12] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, November 2006.

[13] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles – SOSP'11*, 2011.

[14] Miguel Castro and Barbara Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4), November 2002.

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

[16] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *VLDB*, 1(2), August 2008.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing – SoCC '10*, 2010.

[18] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3), August 2013.

[19] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, Joao Paulo, José Pereira, and Ricardo Vilaça. MeT: Workload aware elasticity for NoSQL. In *Proc. of the 8th ACM European Conference on Computer Systems – EuroSys '13*, 2013.

[20] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Elastras: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems*, 38(1), April 2013.

[21] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow.*, 4(8):494–505, May 2011.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of the ACM Symposium on Operating Systems Principles – SOSP'07*, October 2007.

[23] Thibault Dory, Boris Mejas, Peter Van Roy, and Nam-Luc Tran. Measuring elasticity for cloud databases. In *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011.

[24] Cyntia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2), April 1988.

[25] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proc. of the 2011 ACM SIGMOD International Conference on Management of Data – SIGMOD'11*, 2011.

[26] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems*, 30(4), November 2012.

[27] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proc. of the 23rd ACM Symposium on Operating Systems Principles – SOSP'11*, 2011.

[28] J. Hamilton. Observations on errors, corrections, and trust of dependent systems. http://goo.gl/LPTJoO, 2012.

[29] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[30] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proc. of the 2010 USENIX Annual Technical Conference – USENIX ATC'10*, 2010.

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

[32] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about EVE: Execute-verify replication for multi-core servers. In *Proc. of the 10th USENIX Conference on Operating Systems Design and Implementation – OSDI'12*, 2012.

[33] Ioannis Konstantinou, Evangelos Angelou, Christina Boumpouka, Dimitrios Tsoumakos, and Nectarios Koziris. On the elasticity of NoSQL databases over cloud management platforms. In *Proc. of the 20th ACM international conference on Information and knowledge management – CIKM '11*, 2011.

[34] Ramakrishna Kotla and Mike Dahlin. High throughput byzantine fault tolerance. In *Proc. of the 2004 International Conference on Dependable Systems and Networks – DSN'04*, 2004.

[35] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. Napsac: Design and implementation of a power-proportional web cluster. In *Proc. of the 1st ACM Workshop on Green Networking*, 2010.

[36] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2), April 2010.

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

[38] L. H. Le, C. E. Bezerra, and F. Pedone. Dynamic scalable state machine replication. In *Proc. of the 46th IEEE/IFIP International Conference on Dependable Systems and Networks – DSN'16*, June 2016.

[39] William LeFebvre. Cnn. com: Facing a world crisis. In *LISA*, 2001.

[40] Jacob Lorch, Atul Adya, William Bolosky, Ronnie Chaiken, John Douceur, and Jon Howell. The SMART way to migrate replicated stateful services. In *EuroSys*, October 2006.

[41] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kauffman, 1996.

[42] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation – OSDI'08*, October 2008.

[43] Parisa Jalili Marandi, Carlos Eduardo Bezerra, and Fernando Pedone. Rethinking state-machine replication for parallelism. In *Proc. of the 34th IEEE International Conference on Distributed Computing Systems – ICDCS'14*, 2014.

[44] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. of the 24th ACM Symposium on Operating Systems Principles – SOSP '13*, October 2013.

[45] Dushyanth Narayanan, Austin Donnelly, Eno Thereska, Sameh Elnikety, and Antony Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation – OSDI'08*, 2008.

[46] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proc. of the 5th European Conference on Computer Systems – EuroSys '10*, 2010.

[47] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at Facebook. In *NSDI*, April 2013.

[48] Laura Nolan. Managing critical state: Distributed consensus for reliability. In Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall R. Murphy, editors, *Site reliability Engineering*, chapter 23. O'Reilly, 2016.

[49] Brian M. Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. of the 7th Annual ACM Symposium on Principles of Distributed Computing – PODC'88*, 1988.

[50] Diego Ongaro and John Ousterhout. In search for an understandable consensus algorithm. In *Proc. of the USENIX Annual Technical Conference – ATC'14*, June 2014.

[51] Ricardo Padilha and Fernando Pedone. Augustus: Scalable and robust storage for cloud applications. In *Proc. of the 8th ACM European Conference on Computer Systems – EuroSys '13*, April 2013.

[52] Jun Rao, Eugene J. Shenkita, and Sandeep Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *VLDB*, 4(4), 2011.

[53] Hans P. Reiser and Rüdiger Kapitza. Hypervisor-based efficient proactive recovery. In *SRDS*, 2007.
[54] N. Santos and A. Schiper. Achieving high-throughput state machine replication in multi-core systems. In *Proc. of the 33rd IEEE International Conference on Distributed Computing Systems – ICDCS'13*, July 2013.
[55] Fred B. Schneider. Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
[56] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proc. VLDB Endow.*, 7(12):1035–1046, August 2014.
[57] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio P. Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proc. of the USENIX Annual Technical Conference – ATC'12*, 2012.
[58] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3), November 2014.
[59] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The SCADS Director: Scaling a distributed storage system under stringent performance requirements. In *Proc. of the 9th USENIX Conference on File and Stroage Technologies – FAST'11*, 2011.
[60] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *Proc. of USENIX Annual Technical Conference – ATC'12*, June 2012.
[61] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. ZZ and the art of practical BFT execution. In *Proc. of the Sixth Conference on Computer Systems – EuroSys'11*, April 2011.

Andre Nogueira is a researcher of the Department of Computer Science of the Faculty of Sciences of the University of Lisboa, Portugal, and a member of LaSIGE research unit and the Navigators research team. He received his M.Sc in 2008 by the University of Lisbon. His main research interests are dependable and intrusion-tolerant systems. More information at https://sites.google.com/site/andrenogueirasite.

Antonio Casimiro is an Associate Professor at the Department of Informatics of the University of Lisboa Faculty of Sciences (since 1996), and a member of the LaSIGE research unit, where he leads the research line on Timeliness and Adaptation in Dependable Systems. He was adjunct Professor of the Carnegie Mellon Information Networking Institute (2008-2011) and a lecturer at Instituto Superior Técnico of the Technical University of Lisboa (1993-1996). He has a PhD in Informatics (2003) by the University of Lisboa. His research focuses on architectures, fault tolerance and adaptation in distributed and real-time embedded systems, with applications on autonomous and cooperative vehicles and on critical infrastructures monitoring. More information at http://www.di.fc.ul.pt/~casim.

Alysson Bessani is an Associate Professor of the Department of Computer Science of the Faculty of Sciences of the University of Lisboa, Portugal, and a member of LaSIGE research unit and the Navigators research team. He holds a PhD in Electrical Engineering from Santa Catarina Federal University, Brazil (2006), and was a visiting professor at Carnegie Mellow University (2010), and a visiting researcher at Microsoft Research Cambridge (2014). His main interests are distributed algorithms, Byzantine fault tolerance, coordination, and security monitoring. More information at http://www.di.fc.ul.pt/~bessani