Scalable and Decoupled Logging for State Machine Replication

Luiz Gustavo C. Xavier[1,3], **Fernando Luís Dotti**[2], **Cristina Meinhardt**[1,3], **Odorico M. Mendizabal**[1,3]

[1]Centro de Ciências Computacionais
Universidade Federal do Rio Grande (FURG)

[2]Faculdade de Informática
Pontíficia Universidade Católica do Rio Grande do Sul (PUCRS)

[3]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC)

xavier.luizgustavo@gmail.com, fernando.dotti@pucrs.br,

{cristina.meinhardt, odorico.mendizabal}@ufsc.br

Abstract. *State Machine Replication (SMR) is a widely used approach for fault tolerance of important services. Support for SMR implementations on shared infrastructures has emerged, allowing wider adoption. However, there are still non-trivial aspects that developers have to handle to build and deploy their dependable services. In this paper we tackle the need for recovery to keep fault-tolerance levels, and propose an approach to: (i) simplify the development of logging; (ii) improve resource sharing in shared infrastructures; (iii) alleviate costs with replication in pay-per-use infrastructures. The central idea is to decouple service execution from logging and offer logging functionality as a service attachable to SMR deployments. Beyond the added simplicity to deploy an SMR, we show that this approach does not penalize performance of replicated services, and that a logging service can scale to look to several applications.*

1. Introduction

With the increasing demand for highly-available and high-throughput systems, the development of well performing distributed fault-tolerant services becomes a challenging task to a wider group of system developers. State Machine Replication (SMR) is a known technique to manage replicated serves [Lamport 1978, Schneider 1990]. In SMR replicas behave as state machines, starting in the same initial state and deterministically executing an identical sequence of client commands. Many dependable systems entrust SMR with their critical data [Marandi et al. 2016].

In [Altinbuken and Sirer 2012, Netto et al. 2017, Pereira et al. 2019, Borges et al. 2019] authors present different contributions to enable the SMR approach to use shared infrastructures, significantly reducing the initial efforts needed to build replicated services. These efforts, along with strong consistency [Herlihy and Wing 1990], that offers replication transparency, significantly help developers to adopt SMR. Nonetheless, the engineering to build fault-tolerant services still has complex aspects, and we advocate that further functional support is needed.

To provide robustness, a replica should be able to recover. Recovery protocols [Bessani et al. 2013, Mendizabal et al. 2017, Kończak et al. 2019] allow a recovering replica to restore a consistent state and catch up with other replicas in the system. A recoverable state is usually built up from a combination of data acquired from log and checkpoints. To keep this information persisted in a stable storage, both logging and checkpointing strategies have to be adopted, influencing the performance of replicas during normal operation. The typical logging in SMR takes place while processing each command: the command is logged, executed and a reply is sent to the client. When considering high-throughput systems logging can account for a considerable portion of the processing time of each command. Also, logs grow with the throughput of the system and thus have to be frequently truncated, increasing checkpoint frequency. This again penalises the replica's performance [Bessani et al. 2013, Mendizabal et al. 2016].

Logging thus represents an aspect of complexity during the development and deployment of replicated services. Therefore in this paper we present a logging service decoupled from the application, with the goal of: (i) simplifying the development of durability techniques; (ii) improving resource sharing in shared infrastructures; (iii) alleviating costs with resource usage in pay-per-use infrastructures. As main technical aspects, we propose this decoupling by: (a) having a specialized log service available in the form of logger processes; (b) including logger processes as part of the replicated service, sharing the same agreement protocol with state machine replicas. By doing so, these light processes can log the same sequence of ordered commands delivered to each replica. Logging of commands is transparently executed by the logger with no need for modifications in the execution of commands. In a shared infrastructure, multiple services could share a single logging service. Besides sharing the same logger, services can start or stop using the logging service at runtime. The simple API and elasticity offered by our decoupled logging service fits well to approaches where dependable services are deployed in shared infrastructures, such as those already mentioned. Also, this approach relieves replicas from logging commands, reducing both the replica's overhead caused by sequential logging operations and the total amount of data stored in persistent storage.

The rest of this paper is organized as follows. Section 2 exposes close related contributions, that focus on cost reduction with durability and state recovery optimizations. Section 3 contains the system model. In Section 4 we briefly discuss about SMR and recovery. Section 5 presents the decoupled logging strategy and some aspects for its implementation. In Section 6 we experimentally evaluate our approach and compare it with traditional logging schemes. Section 7 concludes this work.

2. Related Work

UpRight [Clement et al. 2009] introduces the idea of a *helper process*, a slight deviation of a regular application that asynchronously captures application snapshots. Two similar threads, the primary and the helper, execute in parallel. While the primary processes requests and sends replies to the clients, the helper periodically takes checkpoints. This strategy avoids pausing of execution of new requests during a checkpoint, and minimizes intrusiveness to legacy code. The application decoupling for checkpoint creation adopted by UpRight motivated us to decouple logging from commands execution. Different from UpRight, though, where primary and helper run at the same machine, our strategy allows distributed execution, where logger and application execute in remote machines.

Some approaches benefit from the agreement protocol to restore a consistent state. In [Boichat et al. 2003] authors describe recovery strategies that rely on Paxos to record the log of decided commands. One of the strategies requires Paxos to write data to a stable storage once per decision. Another approach does not use a stable storage, but it requires that a majority of correct processes is permanently under execution. Upon recovery, a replica broadcasts a message indicating that it recovered its state and it does not vote in further decisions. This strong restriction might be unsuitable for practical systems.

Kończak *et al.* [Kończak et al. 2019] also present recovery algorithms for Paxos-based replicated systems. Authors propose three versions of the algorithm (FullSS, ViewSS and EpochSS). Compared with [Boichat et al. 2003], taking weaker and more practical assumptions about the system. FullSS follows the original Paxos definition and the system frequently uses stable storage during normal execution. The ViewSS algorithm extends Paxos with a recovery phase, and requires that the new ballot number is synchronously written to stable storage on leader changes. The EpochSS algorithm requires that every process stores in stable storage an epoch number, incremented every time the process restarts. Thus, the EpochSS protocol requires only one synchronous write to stable storage per fault-free run of a process. Both ViewSS and EpochSS algorithms assume a limit on the number of processes that may crash at the same time. In [Boichat et al. 2003, Kończak et al. 2019] Paxos is customized to provide logging and recovery mechanisms for replicas. Different from their approach, our logging service provides a simple API to service replicas without changes in the underlying protocol.

Some practical implementations of consensus protocols, such as Multi-Ring Paxos [Benz et al. 2014], allow recovering replicas to retrieve the log of commands by querying the sequence of commands previously decided. Through a *relearn* command, a recovery replica proposes *null* values to an interval of already decided instances of consensus. Participants return the real value decided in each of the requested instances. Although providing a very simple API for recovery, this strategy introduces extra costs to the participants of the consensus protocol, since they have to reach a decision for a possibly large sequence of old values while they continue ordering upcoming commands from the service clients. This overhead may temporarily cause a perceptible increase on latency by the clients [Bessani et al. 2013, Mendizabal et al. 2017]. Furthermore, the *relearn* command provided by Multi-Ring Paxos is not typically made available as a regular Paxos command. Then, similar solutions would be unavailable when using another implementation of Paxos, Raft [Ongaro and Ousterhout 2014] or any other agreement protocol.

Corfu [Balakrishnan et al. 2013] is a distributed and shared log that allows client operations to be run on parallel. During operation, Corfu's clients maintain a local projection map that stores references to physical log positions divided into *pages* distributed across a cluster of logging modes. Operations indexed to pages located in distinct nodes run on parallel, improving throughput and scalability. Similar to the Corfu's distributed logging, other hacks and tricks allow the system to use the disk efficiently [Chandra et al. 2007, Rao et al. 2011, Bessani et al. 2013]. For instance, in [Bessani et al. 2013], authors propose the parallel logging, which attempts to postpone and to batch synchronous writes in order to reduce their number and alleviate their latency. Such improvements are orthogonal to our approach and might even be used in the development of our decoupled logging service.

3. System Model

We assume a distributed system composed of interconnected processes. There is a limited number of service replicas defined by the set $S = \{s_1, s_2, ..., s_n\}$, a limited number of logger processes defined by $L = \{l_1, l_2, ..., l_m\}$, and an unbounded set $C = \{c_1, c_2, ...\}$ of client processes.

The system is asynchronous in the sense that communicating processes may not be able to obtain responses to their requests in time. In other words, there is no bound on message delays and on relative process speeds. A process may fail by crash and subsequently recover, but processes cannot suffer from Byzantine faults. Service replicas and logger processes are equipped with volatile memory and stable storage. Upon a crash, a process loses the content of its volatile memory, but the content of its stable storage survives the failure. We assume f faulty service replicas, out of $n_r = 2f + 1$ servers, and k faulty logger processes, out of $n_l = k + 1$ loggers.

Processes communicate by message passing, using either one-to-one or one-to-many communication. One-to-one communication is through primitives send(m) and receive(m), where m is a message. If a sender sends a message enough times, a correct receiver will eventually receive the message. One-to-many communication is based on atomic broadcast, whose main primitives are *broadcast*(m) and *deliver*(i, m), where i refers to the consensus instance in which m was decided. This definition implicitly assumes that atomic broadcast is implemented with a sequence of consensus instances identified by natural numbers (e.g., [Lamport 1998]). This choice is made consciously, based on the fact that introducing the consensus instance in the delivery event, a service replica can easily determine the messages it needs to retrieve upon recovering from a failure. Analogously, a logger process can keep the same sequence of logged messages observed by the message delivery protocol.

Atomic broadcast ensures that (i) if a process broadcasts message m and does not fail, then there is some i from an infinite ordered set of values such that eventually every correct process delivers (i, m); and if a process delivers (i, m), then (ii) all correct processes deliver (i, m), (iii) no process delivers (i, m') for $m \neq m'$, and (iv) some process broadcasts m. Atomic broadcast requires additional system assumptions to be implemented [Chandra and Toueg 1996].

4. State Machine Replication

SMR renders a service fault-tolerant by replicating the server and coordinating the execution of client commands among the replicas [Lamport 1978, Schneider 1990]. The service is defined by a state machine and consists of *state variables* that encode the state machine's state and a set of *commands* that change the service state. The execution of a command may (i) read state variables, (ii) modify state variables, and (iii) produce an output response for the command.

In order to ensure that the execution of a command will result in the same state changes and results at different replicas, commands are *deterministic*, i.e., the changes to the state and the response of a command are a function of the state variables the command reads and the command itself. Therefore, if servers execute commands in the same order, they will produce the same state changes and results after the execution of each command.

SMR provides clients with the abstraction of a highly available service while hiding the existence of multiple replicas. This last aspect is captured by *linearizability*, a consistency criterion [Herlihy and Wing 1990]: a system is linearizable if there is a way to reorder the client commands in a sequence that (i) respects the semantics of the commands, as defined in their sequential specifications, and (ii) respects the real-time ordering of commands across all clients [Attiya and Welch 2004]. In SMR, linearizability can be achieved by having clients atomically broadcast commands and replicas execute commands sequentially in the same order.

4.1. Recovery in SMR

To allow replicas to be recovered after the occurrence of faults, durability strategies must be implemented, such as logging, checkpointing and state transfer. Most SMR implementations follow a similar recovery approach [Boichat et al. 2003, Rao et al. 2011]:

(i) Commands are logged in the order in which they are executed (as part of atomic broadcast) and a reply is only sent to the client after the corresponding command is logged and executed.

(ii) Each replica periodically checkpoints its state to stable storage. Logged commands preceding the moment in which the checkpoint was taken are discarded from the log. Old checkpoints common to all replicas are also discarded.

(iii) Upon recovery, the replica:
  - (iii-a) reenters consensus and discovers the first consensus instance i decided after its recovery;
  - (iii-b) retrieves and installs the latest checkpoint from another replica or from remote storage; and
  - (iii-c) retrieves the log with executed commands that are not reflected in the checkpoint, up to i, and sequentially execute these commands against the state.

  Client commands delivered during recovery are only processed after the recovery procedure is complete.

5. Decoupled Logging

We propose a technique that separates the log handling from the application. The key idea is to have service replicas dealing exclusively with command execution, free from logging overhead, while an independent service logs the sequence of commands processed.

Figure 1 illustrates our strategy, which decouples logging from the application logic by adding logger processes to the system (*Log 1* and *Log 2*). These light processes log the same sequence of ordered commands delivered to each replica, without executing them. Actually, logger processes do not implement the application logic and can be implemented in a generalized way, without entering in details of the service.

In order to ensure a total ordered sequence of commands observed by both replicas and loggers, logger processes participate in the same agreement protocol with state machine replicas. Towards this end, logger processes are associated to the system at initialization, exactly as service replicas do. For instance, for SMR implementations using Paxos-based libraries, for instance [Benz et al. 2014], loggers play the *learners* role. For those implementations based on Raft protocol, they can execute as *non-voters*. Joining

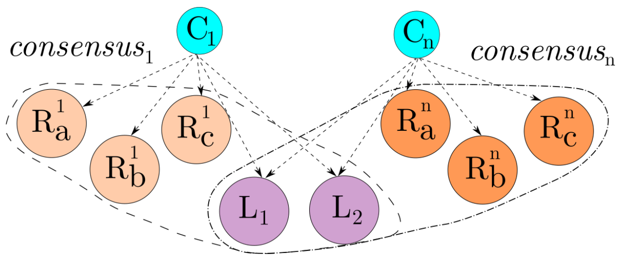


Figure 1. Logging approach for State Machine Replication

loggers to the replicas group is dependent on SMR implementation. Notice, though, that loggers and replicas join to the system in exactly the same way, which simplifies the implementation and deployment of logger processes.

By adding these specialized processes to the system, an extra assumption has to be made. While $n_r = 2f + 1$ service replicas are required to tolerate up to f crashed service replicas, adding logger processes impose $n_l = k + 1$ logger processes to tolerate up to k crashed loggers. Since loggers do not participate in the agreement of commands ordering, it is enough that a single logger is correct.

5.1. Discussion of recovery correctness

Here we argue that a recovery procedure with the proposed decoupled logging reestablishes a consistent replica state. Under the assumption of logger fault-tolerance, a logger will be available to retrieve commands. As logger processes deliver commands in total order, the produced logs follow the execution order. The recovery behavior assumed is as described in Section 4.1, where step (iii-c) is implemented by requesting the logger service for the commands from the last one checkpointed up to the current one. Since we decouple logging from execution, we have to consider that loggers and replicas could have different heights in the computation prefix. Case 1 - if loggers are ahead of replicas execution, the recovering replica will immediately retrieve the interval of missing commands. Case 2 - if replicas are ahead of loggers, then due to loggers fault tolerance and participation in consensus they will eventually reach instance i (the first decided after recovery started) and thus have the interval of commands needed to serve the recovering replica. This could possibly delay but not prevent recovery.

5.2. Performance Considerations

When logging is separated from the application logic some important performance considerations emerge due to decoupling itself and due to the different relative speeds of logger and server replicas.

First, service replicas do not need to log each command. Second, supposing well equipped loggers, for instance running on top of servers with file systems and devices optimized for I/O operations, massive I/O operations could be performed and thus keeping longer logs as well as truncating the log are no longer critical. As a consequence, efforts to frequently take checkpoints and truncate the log become less important. That means service replicas could relax the periodicity in which they checkpoint and request log truncation. As a positive side-effect, reducing checkpoint frequency would directly result in further performance gains at replicas, as discussed in [Mendizabal et al. 2016]. A third important aspect concerns the relative speeds of replica and logger processes.

- If loggers are faster than replicas, and if replicas momentarily do not provide throughput enough to serve incoming requests, then replicas would lag behind loggers. This will not harm recovery, as discussed in Section 5.1. In fact, since this depicts a scenario of saturation at replicas, would be even worse if we had the classic case of replicas logging commands as well.
- If replicas are faster than loggers, and if loggers do not provide throughput enough to log incoming requests, then loggers would lag behind replicas. If such saturation is momentary then eventually the logger will catch up, also as discussed in Section 5.1. In such case a request to retrieve a log interval will suffer some latency and eventually be served when the logger has all requested commands.

  If logger saturation however becomes prominent, we would face a gradual increase in the delay of commands delivered to logger processes, and an increase in the delay to recover of replicas, since a recovering replica would request commands that were not stored by the logger yet, so it has to wait until the logger is able to transfer the whole log. To mitigate these problems, a replica could truncate log commands always it is possible, i.e., right after taking a checkpoint. This way, logger processes would be requested to trim commands they had not processed yet. By doing this, upon receiving commands behind the truncation point, logger processes can simply discard those commands and resume logging of commands after the truncation point. Even such mitigation is possible, the ideal configuration is such that the loggers do not lag behind service replicas.

5.3. Application Programming Interface

Regarding engineering and architectural design, by decoupling logging from application, developers do not need to rewrite applications from scratch or implement optimized persistence routines to maintain the application log. Service replicas interact with loggers by using a very simple API, provided by the logger library and described bellow.

`recover(first, last)` The `recover` routine allows replicas to request an interval of commands from the log. It is usually needed when a recovering replica joins to the system or a new one is started, so it must acquire an interval of missed commands to catch up with other replicas. `first` and `last` represent endpoints of a closed interval, where the `first` corresponds to the first command to be processed after installing a snapshot, or the first command when no checkpoint is available. The `last` corresponds to the last command that has to be processed, once the next command can be delivered by the agreement protocol.

`truncate(instance)` The truncate routine allows a replica to inform a safe instance which can be used by loggers to trim a prefix of commands from the log up to the command given by `instance`. This routine is used when a majority of correct replicas has successfully saved a snapshot of the service state containing all updates up to the command of instance `instance`. The truncate routine will effectively truncate the log only when more than $(n-1)/2$ replicas ask the logger for truncating. In this case, the lowest informed value of `instance` will be used to truncate the log. This quorum of replicas is needed to avoid a single replica checkpoints, ask the logger for truncating, and fails. Notice that such case would lead the system to a unrecoverable state, where log discarded commands that cannot be recovered from a snapshot.

6. Experimental evaluation

This section presents an experimental evaluation of our technique. Three main aspects are discussed:

1. Evaluating the impact of performance on replicas during normal execution. On the one hand, separating logging from the application alleviates the cost with routines for data persistence. On the other hand, once loggers are added to the group of replicas, some performance degradation might be observed by the consensus protocol due to the increased message density. The analysis compares the throughput of *application-level logging* (*i.e.*, the typical logging approach) and *decoupled logging* (*i.e.*, the proposed approach);
2. Evaluating the performance behavior of loggers w.r.t. replicas. This is observed by comparing application and logger throughput, where a lower throughput on logging commands indicates the logger is slowing down. This situation must be avoided, since it would delay recovery or the addition of replicas in the system.
3. Checking the feasibility of sharing the decoupled logging service by multiple applications. The objective of this study is to evaluate how the distributed log service scales with the number of applications.

6.1. Workloads and Distributed Applications

We implemented application prototypes that exercise both *CPU* and *I/O-bound* workloads. This way, we can observe the impact of commands logging in scenarios with low and high competition for disk writes. By using the decoupled logging service in I/O intense scenarios, it is expected a lower contention to the disk since the application does not need to store commands in a log file. Two distinct distributed applications were implemented to measure the efficiency of log decoupling, in terms of throughput overhead and latency impact observed by clients.[1] They were developed using Go programming language and implement a SMR model using Hashicorp's Raft implementation [Hashicorp 2014] as consensus layer. Serialization of structured data is implemented with protocol buffers.

kvstore implements a key-value store, where user's data is kept in-memory. Client may read and write data by invoking `get(k)` and `set(k,v)` operations. The key `k` is represented by a integer number and the value `v` is a byte array with configurable size. In our experiments, 1,000,000 distinct keys were used, and values size were set to 1,024.

diskstorage implements a directory service. Directory entries are kept in a persistent storage. This application induces an I/O-intense behavior. Application state is fully represented by a 1GB file, which is modified by `read(offset)` and `write(offset,v)` requests, where `offset` describes a record in the directory and `v` is a byte array with a fixed size. In our experiments values size are set to 1,024. Disk updates are synchronous commands.

6.2. Experiments

For each application, we execute (i) a baseline application, where the logging service is disabled; (ii) a version where besides processing client requests, the application is responsible for logging commands (the typical logging approach); (iii) application running with decoupled logging service, where commands are logged by remote logger processes.

[1]Applications available at `github.com/Lz-Gustavo/raft-demo/tree/v1.2`

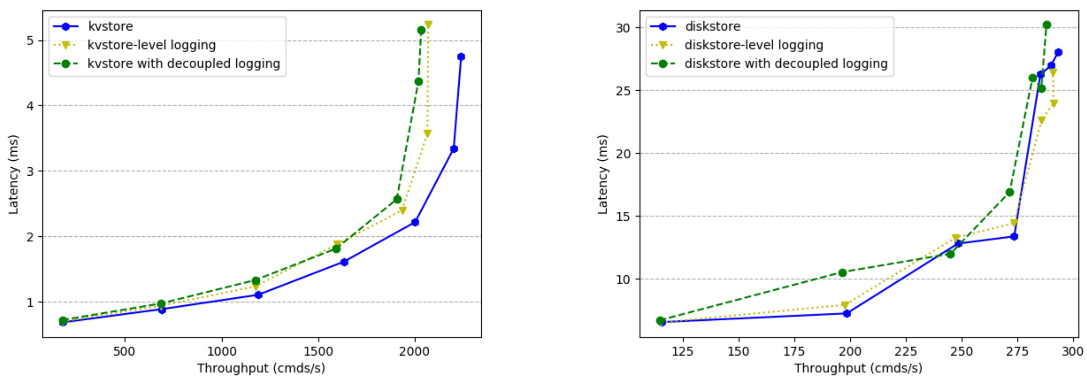
Application clients are simulated by load generators, which initialize a set of client threads. In order to avoid contention at the client side, every load generator runs at most 6 client threads. Every client thread randomly generates a command (read or write operation) and broadcasts it for replicas execution. When the client thread receives the response for a submitted command, it randomly waits for a maximum 10ms, generates a new command and repeats the sending procedure.

6.3. Environment and configuration

Experiments were executed adopting a topology of bare-metal Dell PowerEdge 1435 nodes, equipped with 2x Dual-Core AMD Opteron(tm) processor running at 2GHz and 4 GB of main memory (DDR2 667MHz SDRAM), and a storage controller SATA II, 7.2k RPM, 500 GB with a 16 MB buffer. All nodes are connected to an HP ProCurve switch 2920–48G gigabit. All nodes run Ubuntu 18.04 LTS Linux. Our prototype was set up to tolerate one service failure, and one logger failure, requiring three application replicas and two logger processes, all running in independent nodes.

6.4. Results

Figures 2 shows the throughput versus latency graph for the *kvstore* and *diskstorage* applications both with value size set to 1,024 bytes. The number of emulated clients was gradually increased to generate each point in the graphics. As expected, the version of application without logging presents the best performance, reaching a maximum throughput in both applications (around 2,500 commands/s for *kvstore* and 280 commands/s for *diskstore*) and lower latency values. Figure 2(a) describes the execution of the *kvstore* application, addressing a typical CPU intensive workload. As observed, the application-level logging, where application is in charge of registering commands in the persistent device, presents a slightly better performance when compared to the decoupled logging strategy. The addition of logger processes in the consensus protocol might incur in extra costs to the decoupled logging approach.



(a) key-value store (b) disk storage

Figure 2. Throughput versus latency of 1k-byte commands.

When running I/O-bound applications, the logging of commands increases competition for disk access between commands execution and logging. As depicted by Figure 2(b), by running *diskstorage*, application performance is severely impacted, reaching a saturation point when throughput approaches 280 commands per second. However, differences among logging strategies are less perceptible. Even the configuration without logging reaches a throughput very similar to those achieved when log is enabled.

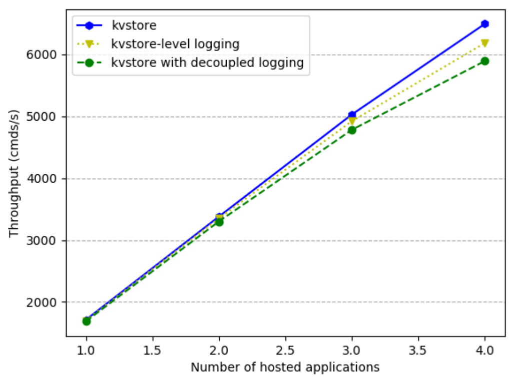
According to our results, the addition of nodes to execute logger processes may cause a small reduction in the maximum observed throughput, especially when workload is CPU intensive. This cost is negligible for I/O-bound applications. Although there is apparently no direct benefit in using the decoupled logging approach for a single application, sharing a single logging service among multiple applications may contribute to better use of resources, thereby saving costs, especially when services are deployed in *pay-per-use* infrastructures.

In order to evaluate the impact of logging in shared infrastructure, we run multiple instances of *kvstore* and *diskstore* applications in the same nodes. More precisely, we choose groups of 3 nodes to host replicas of applications and 2 nodes to run the replicated logging service. At most 4 shared applications are allocated per physical machine, while a single pair of machines is used to deploy the logging service. The limit of 4 shared applications was imposed since it is the maximum number of applications hosted in server nodes before degradation is observed. However, the pair of machines used to deploy logging service is capable to continue serving more applications. Each hosted application is submitted to a workload of approximately 70% of their maximum load.

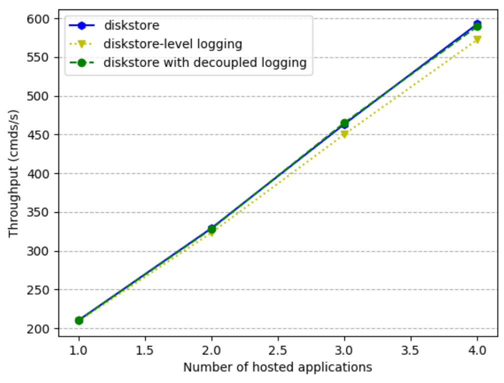
Figure 3 depicts the throughput variation according to the number of independent applications running in the same group of servers. The x-axis indicates the number of hosted applications, while the y-axis shows the cumulative throughput, *i.e.*, the sum of the average throughput observed by each application individually. As expected, the highest cumulative throughput is observed when logging is disabled. For CPU-intensive applications (see Figure 3 (a)), as the number of hosted applications increases, both application-level and decoupled logging experienced a decrease in the throughput growth rate. However, when evaluating the impact of logging in a shared infrastructure running *diskstore* application (see Figure 3 (b)), the throughput of application without logging and application using our decoupled logger service are practically the same. A small reduction on the throughput growth rate is observed when application is responsible for logging commands. This performance reduction is a consequence of a higher competition to the disk caused by both commands execution and logging of commands.

In order to evaluate the scalability of our approach, we gradually increased the number of applications served by the decoupled logger service. In this experiment, we limited a maximum of 3 applications per group of replicas. Figure 4 shows the cumulative throughput of *kvstore* with decoupled logging and the cumulative throughput of the logger itself. We increased the number of application up to 9, while both application and logger presented a very similar throughput. That means the logger is capable to handle this load. Unfortunately, we did not have extra nodes to continue increasing the number of applications, which limited the scalability graph to 9 applications. With such load, we could not identify the saturation point of the logger (we expected a slow decay at throughput curve after a certain number of running applications, indicating the logger is slowing down).

As a way to redress the lack of resources in the previous experiment, and to strengthen the scalability study, we evaluated the maximum throughput achieved by the



(a) key-value store



(b) disk storage

Figure 3. Cumulative throughput of the logger shared by several applications.

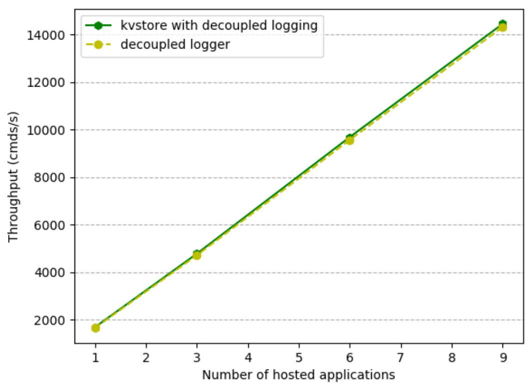


Figure 4. Scalability analysis of decoupled logging service.

logging routine and by the consensus protocol separately. First, we reproduce the workload of *kvstore* and record the generated commands locally, with the absence of network and ordering costs. The average throughput observed for logging of commands is around 46,000 commands/s. With 9 applications the logger was subject to 14,000 commands/s, which represents 30% of the maximum logging throughput. In order to estimate the maximum throughout of commands delivered, we implemented a Raft program that continuously proposes commands and replicas simply discard the delivered commands. By running an intense workload, we observed a delivery rate of 3,200 commands/s. Previous tests were generating an average throughput of 2,200 commands/s, which means their throughout was not limited by the consensus protocol. These benchmarks cannot replace a detailed analysis of scalability, but they give some evidences that the logger would be capable to handle additional applications before degrading performance. Future work should extend the scalability analysis to provide more accurate information.

By allowing multiple applications to share a logging service, our approach reduces the number of nodes in charge of writing logging information into the stable storage. As discussed in Section 3, $n_r = 2f+1$ servers, and $n_l = k+1$ loggers are required to tolerate f and k faults, respectively. Equations 1 and 2 depict the monetary costs of the traditional and decoupled logging strategies. These equations involve the following variables:

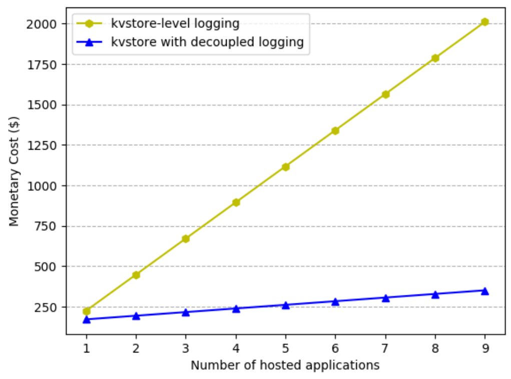
Symbol	Meaning	Value for *kvstore*	Value for *diskstorage*
n	# of applications using same logger	1..9	
f	replicas that may fail	1	
k	loggers that may fail	1	
d_a	disk usage by application / min.	0 MB	50 MB
d_l	disk usage by logger / min.	59 MB	10 MB
ϕ	fee per byte used ($)	$\phi = 0.025/10^9$ bytes	
ϵ	cost for a single machine ($ / month)	$\epsilon = 0.0104 \times 720$	

$$C_{app-level} = \phi\{n[(2f+1) \times (d_a + d_l)]\} + \epsilon[n(2f+1)] \tag{1}$$

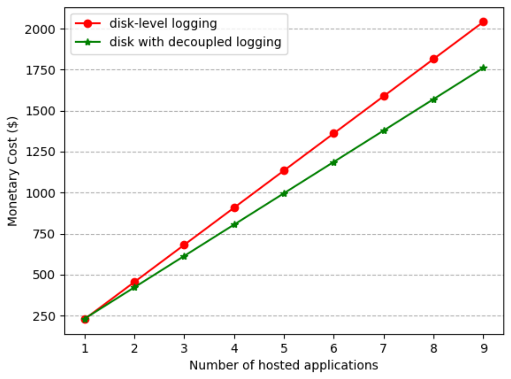
$$C_{decoup-log} = \phi\{n[(2f+1) \times d_a] + [(k+1) \times d_l]\} + \epsilon[n(2f+1) + (k+1)] \tag{2}$$

In order to estimate the monetary impact with the decoupled logging, we adopted the current pricing policies charged by *Amazon Web Services* [AWS 2019], considering `sc1` EBS volumes and `t3.micro` EC2 instances for each server. Then, we set: $\phi = 0.025/10^9$, which represents a fee of $0.025 per GB used; and $\epsilon = 0.0104 \times 720$, which represents the cost of `t3.micro` nodes per month. Values of d_a and d_l were obtained by monitoring applications configured as described by scenarios shown in Figure 3. Every minute, we observed $d_a = 0$ and $d_l \approx 59$ MB for *kvstore*, and $d_a \approx 50$ MB and $d_l \approx 10$ MB for *diskstorage*. *kvstore* has none d_a usage due to absence of I/O operations, and a greater d_l because of its higher throughput. Monthly costs for both d_a and d_l were obtained by multiplying their values by 60 (minutes), by 24 (hours), and by 30 (days).

Therefore, we can observe a monthly cost reduction of 23.28% for *kvstore* and an increase of 1.4% for *diskstorage*. By extending for nine applications, the monetary cost is reduced to 82.54% and 13.72%, which can represent important savings on *pay-per-use* infrastructures. As depicted in Figure 5, cost reduction is observed on *kvstore* even with a single application under execution, whereas for *diskstorage* a slower gain is observed with three or more applications. As can be seen, the benefits are proportional to d_l and n. This inquires that the greater is the application's throughput, the greater will be the monetary savings by sharing a decoupled log with different applications.



(a) key-value store



(b) disk storage

Figure 5. Monthly costs of decoupled logging shared by multiple applications.

Finally, we evaluate the impact of decoupled logging upon recovery of replicas. Towards this end, we start a new replica during a test execution, so it retrieves the complete log of commands by issuing a `recover` request and sequentially processes every command in the log. From this moment on, the replica is recovered and able to execute new commands. In order to compare traditional and decoupled logging strategies, we evaluate the recovery time and application throughput for both configurations. The new replica is started after 3 minutes of test execution. At this time, log contains around 380,000 commands, which corresponds to approximately 195 MB. Both techniques presented a similar behavior. The time taken to transfer the log remains close to 2.7 seconds, while the processing time was fixed in 1.65 seconds, which results in approximately 4.4 seconds to recovery the replica. Neither logging techniques caused throughput fluctuations in application during recovery.

7. Concluding Remarks

This paper presents a logging service for SMR decoupled from application. The general idea is to add light processes, called loggers, to the set of replicas in the system. Loggers are in charge of maintain and retrieve the log of commands processed by the replicas. The paper describes the decoupled logging approach, discusses recovery correctness and some performance implications of this approach.

From a programming perspective, by using this service, developers do not need to rewrite applications from scratch or implement optimized persistence routines to maintain the application log. The logger service provides a very simple API, allowing service replicas to retrieve the log at recovering or truncate the log to eliminate a prefix of commands no longer needed to restore a consistent state.

From a performance perspective, besides alleviating the overhead of service replicas with I/O operations, loggers can support multiple services at the same time. A sustainable use of resources is especially attractive to reduce costs of service providers, such as cloud providers. In this sense, the decoupled logging service becomes a good alternative to dependable services running at shared environments, such as cloud infrastructures.

Future steps in this work include moving the logging service to a component in a cloud service provider. Replicated services running into the cloud should rely on our logger service to keep track of commands processed by replicas and to retrieve a log prefix at recovery. Another improvement would be to apply specific optimizations and test the shared logger using specialized hardware for I/O.

References

Altinbuken, D. and Sirer, E. G. (2012). Commodifying replicated state machines with openreplica. available at `http://openreplica.org/static/papers/OpenReplica.pdf`.

Attiya, H. and Welch, J. (2004). *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience.

AWS (2019). AWS EBS and EC2 pricing values. `https://aws.amazon.com/`.

Balakrishnan, M., Malkhi, D., Davis, J. D., Prabhakaran, V., Wei, M., and Wobber, T. (2013). Corfu: A distributed shared log. *ACM TOCS*, 31(4):1–24.

Benz, S., Marandi, P. J., Pedone, F., and Garbinato, B. (2014). Building global and scalable systems with atomic multicast. In *ACM MIDDLEWARE 2014*.

Bessani, A., Santos, M., Felix, J., Neves, N., and Correia, M. (2013). On the efficiency of durable state machine replication. In *USENIX ATC 2013*.

Boichat, R., Dutta, P., Frølund, S., and Guerraoui, R. (2003). Deconstructing paxos. *ACM SIGACT 2003*.

Borges, F., Pacheco, L., Alchieri, E., Caetano, M. F., and Solis, P. (2019). Transparent state machine replication for kubernetes. In *IEE AINA 2019*.

Chandra, T. D., Griesemer, R., and Redstone, J. (2007). Paxos made live: an engineering perspective. In *ACM SIGACT 2007*. ACM.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riche, T. (2009). Upright cluster services. In *ACM SIGOPS 2009*.

Hashicorp (2014). Raft GitHub repository. `https://github.com/hashicorp/raft`.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS 1990*, pages 463–492.

Kończak, J. Z., Wojciechowski, P. T., Santos, N., Żurkowski, T., and Schiper, A. (2019). Recovery algorithms for paxos-based state machine replication. *IEEE TDSC 2019*.

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

Lamport, L. (1998). The part-time parliament. *ACM TOCS*, 16(2):133–169.

Marandi, P. J., Gkantsidis, C., Junqueira, F., and Narayanan, D. (2016). Filo: Consolidated consensus as a cloud service. In *USENIX ATC 2016*.

Mendizabal, O. M., Dotti, F. L., and Pedone, F. (2016). Analysis of checkpointing overhead in parallel state machine replication. In *ACM SAC/DADS 2016*.

Mendizabal, O. M., Dotti, F. L., and Pedone, F. (2017). High performance recovery for parallel state machine replication. In *IEEE ICDCS 2017*.

Netto, H. V., Lung, L. C., Correia, M., Luiz, A. F., and de Souza, L. M. S. (2017). State machine replication in containers managed by kubernetes. *JSA*, 73:53–59.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *USENIX ATC 2014*.

Pereira, P. M., Dotti, F. L., Meinhardt, C., and Mendizabal, O. M. (2019). A library for services transparent replication. In *ACM SAC/DADS 2019*.

Rao, J., Shekita, E. J., and Tata, S. (2011). Using paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM CSUR 1990*.