Systems Support For Efficient State-Machine Replication

Gerhard Habiger
gerhard.habiger@uni-ulm.de
Ulm University

Franz J. Hauck
franz.hauck@uni-ulm.de
Ulm University

ABSTRACT

State-Machine Replication (SMR) is a well known approach for the deployment of highly fault-tolerant services. Recent research has focused on efficiency improvements, performance optimisation and novel approaches to underlying concepts of SMR, such as consensus with trusted components, dynamic weights for quorums, or parallelisation of application code. To increase adoption of SMR as a basic fault-tolerance technique, we see the need to improve the current state of the art of SMR even further, and provide four specific ways in which our research contributes to this goal. In particular, we present two approaches which make the development and deployment of SMR services both easier and more efficient, and talk about two further areas of improvement concerning internal mechanisms of common SMR architectures.

The goal of this paper is to provide our current understanding of important issues of current SMR systems as well as to outline possible future solutions to them.

KEYWORDS

State-Machine Replication, Fault Tolerance, Deterministic Multithreading, Deterministic Checkpointing, Consensus

1 INTRODUCTION

State-Machine Replication (SMR) is a well-known approach to achieve fault-tolerant services [17]. Generally, an SMR service is modelled as a deterministic state machine. Every request (input) is deterministically and reliably delivered to all replicas, each processing every incoming request in a deterministic way. All correct replicated state machines will have the same deterministic and consistent state even if a defined number of replicas may fail. SMR can be used with a crash-stop or crash-recovery model. Additionally, SMR is one of the few approaches capable of dealing with Byzantine failures.

Due to its high cost for this high level of reliability, SMR is usually used for critical services, e.g. coordination and locking services like Zookeeper [8], etcd [18] and Chubby [4]. However, the strict determinism requirements of SMR makes development of such systems complicated. Therefore special systems software, i.e. mature frameworks and middleware systems are desirable for the development of SMR-based applications. Only very few such frameworks are currently available and mature enough for serious deployment. Interestingly enough, Zookeeper and etcd do not rely on frameworks, but deeply integrate SMR mechanisms with application code—mainly for performance reasons. This has caused problems in both projects in the past [10, 14], which demonstrates how intricately all parts of an SMR system have to fit together in order for the whole system to work correctly.

1.1 Motivation and Structure

It is our goal to make the development and deployment of SMR applications more accessible by providing suitable systems software which aids developers when designing, implementing and deploying generic applications utilising SMR. This position paper will outline past and current research activities of our team to support efficient and accessible SMR. In particular, the following topics will be expanded upon in the remaining chapters of this paper:

- Whereas many systems execute requests sequentially inside a replica, we allow concurrent execution based on deterministic multithreading. We designed sophisticated and configurable deterministic scheduling algorithms, which allow for parallel processing of requests without the need to define *classes* or *types* of requests beforehand, as is proposed in recent efficient SMR parallelisation approaches.
- SMR systems often face variable load, e.g. during different times of day or depending on the weekday or current events. We designed and implemented a system capable of vertically scaling CPU cores during runtime, which saves resources during low-load scenarios without sacrificing peak load capabilities. Our reconfigurable schedulers allow for even more scaling decisions, finally leading to fully self-optimising SMR systems which adapt to the current application, load and client profiles.
- Input distribution to all replicas typically relies on fault-tolerant consensus algorithms, e.g. Multipaxos. Consensus algorithm that exploit trusted components improve efficiency for Byzantine faults, because latency and the number of necessary messages can be reduced. One promising of such algorithms is EBAWA [19]. We identified and repaired a number of liveness and safety bugs of EBAWA, and even improved it further to be more efficient.
- Concurrent execution of requests makes consistent checkpointing in replicas much more difficult due to the uncertainty of changed data in concurrent threads. However, checkpointing is required to bootstrap new and recover old replicas, and to prune log data that is required for the consensus algorithm. We implemented several approaches to deterministically checkpoint replicas even while concurrent threads execute requests.

With these contributions, we aim to make SMR more usable and available to a wider audience. Thus it becomes a basic technology for critical systems requiring high levels of fault tolerance.

FBSYS '19, November 21–22, 2019, Osnabrück, Germany


https://doi.org/10.18420/fbsys2019-04

We want to emphasise that systems support for efficient SMR deployments is possible, so that impementation and optimisation details are hidden in a middleware or framework layer. Further we will show that there are many difficult open issues that are worth to be solved to attract more users to SMR.

2 RESEARCH AND GOALS

For the main motivations mentioned in our introduction, we will present our current and past research, as well as the open problems we are planning to solve.

2.1 Deterministic Scheduling

With SMR systems requiring deterministic execution of requests in order to stay consistent across all replicas, the easiest way of implementing such systems keeps all application code in a single thread and makes sure no explicit randomness is introduced by calls to PRNGs or similar. Multicore processors have long since taken over the world and there are significant performance benefits to be gained by parallelising applications. One approach to parallelise deterministic applications as required by SMR is to specifically classify requests into groups such as "conflicting" and "non-conflicting" or similar. Recent research shows that this approach manages to yield improved performance [1, 11]. Most of these solutions, however, require the explicit classification of requests by the application developer during the development process, which requires the developer to not only be aware of this problem, but also to learn and think in terms of the specific parallelisation solution and its proposed classification scheme. Other solutions speculatively execute requests which are likely to not interfere, but have to sacrifice performance whenever this speculation goes wrong [9].

In contrast, a deterministic scheduler for multithreading is application-agnostic and treats all requests equally, only requiring developers to utilise well-known synchronisation techniques like Mutexes. Any execution of parallelised applications on a deterministic scheduler will always yield the same results, given a correct parallel program with correctly lock-protected shared data regions.

With MAT [16] and UDS [7] we designed two sophisticated and configurable scheduling algorithms. UDS in particular is flexible enough to simulate other existing schedulers like MAT, PDS [2] and Kendo [15]. Since different configurations of the scheduler are optimal for different application types, UDS has the built-in capability to deterministically adapt configuration parameters during runtime, so that an SMR system can adapt itself to a wide variety of load situations on the fly.

It is already known for a while that there is no one-fits-all solution when it comes to scheduling strategies [5]. Application behaviour, request load, number of available cores and the scheduling algorithm and its configuration heavily interfere. Several problems like the so-called round-filling delay and unbalanced execution phases of concurrent threads may reduce concurrency. In worst case, scheduling degrades to sequential execution providing no benefits at all. However, clever configurations of a flexible scheduler like UDS can prevent this. Our current efforts are concentrated on measuring the effects of different UDS configurations on common application-load profiles of SMR systems. We will show the benefits of this approach, both in regard to the improved performance compared to single-threaded applications, as well as for ease of development of SMR-capable parallelised programs.

2.2 Self-Optimisation

Since load on a system usually fluctuates over time, it would be desirable to have an SMR system optimising itself according to current load and resource utilisation. Optimisation targets can for example be vertically scalable hardware resources like CPU cores, RAM or network interface speed, or system-specific internal parameters like queue lengths inside the consensus algorithm or scheduler specific parameters.

Conceptually speaking, in order to react to changes in load, a system first needs to learn about changes in resource usage by monitoring. However, since an SMR systems consist of multiple replicas with possibly different hardware, and are often even deployed in a Byzantine failure setting, usual monitoring solutions would yield vastly inconsistent results for different machines, resulting in indeterministic reconfiguration and broken systems.

In our research prototypes based on BFT-SMaRt [3], we moved monitoring inside the replicas and separated optimisation targets into non-deterministic (e.g. CPU cores) and deterministic (e.g. scheduler configuration). For non-deterministic targets, we can simply monitor each replica's current resources (e.g. CPU load) and decide on scaling actions locally. For deterministic targets however, we reduced the monitored resources to a single value which is frequently and deterministically determined between all replicas. Based on this value, reconfiguration decisions can be made locally and will be consistent across all replicas.

In a recent paper, we showed that for a properly parallelised application, the optimal configuration for different load levels depends on the current request rate, the request execution time, and—among others—the number of available cores. Scaling cores has no effect on the determinism of the SMR cluster, so each machine can freely choose when and how to scale cores depending on its locally monitored values. In [6], we managed to scale the number of cores dynamically at runtime to save costs and always meet the current demand.

We also showed that reconfiguring our deterministic scheduler can improve performance for different application types and loads compared to static configurations like in previous scheduling algorithms. In contrast to scaling CPU cores, however, reconfiguring UDS impacts determinism, so we decide about scheduler-reconfigurations using a single value derived from the current request rate in the system, which is consistent across all replicas.

Assembling these parts into one dynamic SMR system would yield a fully self-optimising platform, which adapts to current resource utilisation and load based on deterministic and non-deterministic input values. The benefits of such a system are clear, as they allow developers and administrators to focus on the performance of the application itself, instead of fine-tuning system parameters. Additionally, such a system can save significant resources, which translates into saved costs in the right setting.

We are currently working on making our prototypes smarter with respect to their optimisation strategies and plan to publish our results next year.

2.3 Consensus

Since all replicas in an SMR system require the same totally ordered input, oftentimes consensus algorithms which are usually similar to Multipaxos are employed to order incoming requests. Consensus is expensive and requires a lot of communication overhead before a decision is reached. Deploying a trusted component reduces the number of interaction rounds in case of Byzantine failure models as well as the number of necessary messages. The main reason for these reductions is that replicas can no longer cheat on messages, as the trusted component will sign the message and attach a trusted sequence number before signing. Thus a replica is no longer able to send additional messages with its sender ID. Instead replicas always have to be able to show all their messages to other replicas to demonstrate that they were not cheating.

One promising consensus protocol that deploys such a component is EBAWA [19]. EBAWA uses a rotating leader principle, which reduces the impact of malicious nodes that try to behave normal but try to slow down the system as much as possible.

We found out that EBAWA contains multiple liveness and safety bugs, e.g. a situation where a fallen-behind replica can never catch up, and another where replicas diverge due to indeterministic assumptions about the so-called black list.

We fixed EBAWA's design and developed an improved version, which is even faster. The orginal EBAWA algorithm has more or less sequential decisions, whereas our improvement allows concurrent decisions up to an α window as proposed by Lamport [12]. We further modelled EBAWA and our improved version in PlusCal [13] in order to verify the found bugs and to validate for bug-freeness.

2.4 Checkpointing

Lastly, for efficiently running a cluster of replicas in the real world, checkpointing is required not only to enable restarting replicas without having to re-run all requests ever seen by the SMR system, but also to prune logs usually created and required by the consensus algorithm. Further, during consensus a replica may fall behind due to network latencies or performance problems. If this replica is too far behind, it is better to load a checkpoint to catch up instead of trying to execute all missing requests.

Checkpointing becomes incomparably harder for multithreaded systems, even when using simple stop-the-world approaches, for example due to inherent problems with the wait/notify synchronisation of threads or thanks to liveness issues when blocking threads before a checkpoint while others are waiting for their input.

For our internal parallelised prototype SMR system, we worked on enabling checkpointing with different snapshotting approaches. We have a working system which in its simplest form utilises a stop-the-world approach, which waits for all current threads to finish running before taking a checkpoint.

Future work will have to focus on more efficient snapshotting approaches, which could for example be capable of doing rolling snapshots during the execution of parallel threads.

3 CONCLUSION

In the last sections, we presented our research and focus on the idea of a self-optimising, easy-to-use SMR system. We think that Byzantine fault-tolerance should be made as accessible as possible. While frameworks like BFT-SMaRt represented a huge step in that direction, the inherent single-threaded execution and static configuration lack the required dynamicity to satisfy current hardware designs and variable loads of real world applications.

It is our hope that we can contribute to solving open problems like the easy development of deterministic concurrent applications or fully autonomous self-optimisation of highly parametrised platforms, and extend existing systems software by our newly developed mechanisms to better support fault-tolerant applications. Thus, we can make SMR more accessible to a wider audience, increasing its adoption and recognition as one of the most capable fault-tolerance mechanisms available.

ACKNOWLEDGMENTS

This material is based upon work supported by the DFG under Grant No. HA 2207/10–1 and 10–2.

REFERENCES

[1] E. Alchieri, F. Dotti, O. M. Mendizabal, and F. Pedone. 2017. Reconfiguring Parallel State Machine Replication. In *Proc. of the IEEE 36th Symp. on Rel. Distr. Sys. (SRDS)*. 104–113. https://doi.org/10.1109/SRDS.2017.23

[2] C. Basile, Z. Kalbarczyk, and R. K. Iyer. 2006. Active Replication of Multithreaded Applications. *IEEE Trans. Parallel Distrib. Syst.* 17, 5 (May 2006), 448–465.

[3] A. Bessani, J. Sousa, and E. E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *Proc. of the 44th Int. Conf. on Dep. Sys. and Netw. (DSN)*. 355–362.

[4] M. Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *7th Symp. on Oper. Sys. Design and Impl. (OSDI*. USENIX Assoc., Berkeley, CA, USA, 335–350. http://dl.acm.org/citation.cfm?id=1298455.1298487

[5] J. Domaschka, F. J. Hauck, T. Bestfleisch, H. P. Reiser, and R. Kapitza. 2008. Multithreading Strategies for Replicated Objects. In *ACM/IFIP/USENIX 9th Int. Middlew. Conf.*

[6] G. Habiger, F. J. Hauck, J. Köstler, and H. P. Reiser. 2018. Resource-Efficient State-Machine Replication with Multithreading and Vertical Scaling. In *2018 14th European Dependable Computing Conference (EDCC)*. 87–94. https://doi.org/10.1109/EDCC.2018.00024

[7] F. J. Hauck, G. Habiger, and J. Domaschka. 2016. UDS: A Novel and Flexible Scheduling Algorithm for Deterministic Multithreading. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*. 177–186. https://doi.org/10.1109/SRDS.2016.030

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=1855840.1855851

[9] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. 2012. All about Eve: execute-verify replication for multi-core servers. In *Proc. of the 10th USENIX Symp. on Oper. Sys. Des. and Impl. (OSDI)*. 237–250.

[10] K. Kingsbury. 2014. Jepsen: etcd and Consul. https://aphyr.com/posts/316-call-me-maybe-etcd-and-consul.

[11] R. Kotla and M. Dahlin. 2004. High throughput Byzantine fault tolerance. In *Proc. of the Int. Conf. on Dep. Sys. and Netw (DSN)*. IEEE, 575–584.

[12] L. Lamport. 2001. Paxos made simple. *SIGACT News* 32, 4 (2001).

[13] L. Lamport. 2009. The PlusCal Algorithm Language. In *Theoretical Aspects of Computing - ICTAC 2009 (LNCS)*, Vol. 5684. Springer Berlin Heidelberg, Berlin, Heidelberg, 36–60.

[14] A. Medeiros. 2012. ZooKeeper's atomic broadcast protocol: Theory and practice.

[15] M. Olszewski, J. Ansel, and S. Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *Proc. of the 14th Int. Conf. on Arch. Support for Progr. Lang. and Oper. Sys. (ASPLOS)*. ACM, 97–108.

[16] H.P. Reiser, J. Domaschka, F.J. Hauck, R. Kapitza, and W. Schroder-Preikschat. 2006. Consistent Replication of Multithreaded Distributed Objects. In *Reliable Distributed Systems, 2006. SRDS '06. 25th IEEE Symposium on*. 257–266.

[17] F. B. Schneider. 1990. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 22, 4 (Dec. 1990), 299–319. https://doi.org/10.1145/98163.98167

[18] The Cloud Native Computing Foundation. 2019. etcd Homepage. https://etcd.io/.

[19] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. 2010. EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*. 10–19. https://doi.org/10.1109/HASE.2010.19