Storyboard: Optimistic Deterministic Multithreading

Rüdiger Kapitza, Matthias Schunter, and Christian Cachin
IBM Research - Zurich
{rka,mts,cca}@zurich.ibm.com

Klaus Stengel and Tobias Distler
Friedrich-Alexander University Erlangen-Nuremberg
{stengel,distler}@cs.fau.de

Abstract

State-machine replication is a general approach to address the increasing importance of network-based services by improving their availability and reliability via replicated execution. If a service is deterministic, multiple replicas will produce the same results, and faults can be tolerated by means of agreement protocols.

Unfortunately, real-life services are often not deterministic. One major source of non-determinism is multi-threaded execution with shared data access in which the thread execution order is determined by the run-time system and the outcome may depend on which thread accesses data first.

We present *Storyboard*, an approach that ensures deterministic execution of multi-threaded programs. Storyboard achieves this by utilizing application-specific knowledge to minimize costly inter-replica coordination and to exploit concurrency in a similar way as non-deterministic execution. This is accomplished by making a forecast for a likely execution path, provided as an ordered sequence of locks that protect critical sections. If this forecast is correct, a request is executed in parallel to other running requests without further actions. Only in case of an incorrect forecast will an alternative execution path be resolved by inter-replica coordination.

1 Introduction

Network-based services have constantly increased in importance for our every-day life. Accordingly, there is a trend to offer these services 24/7 by making them tolerate hard- and software faults ranging from plain crashes to arbitrary faults. Besides providing a well-functioning service, this requires additional measures such as state-machine replication [14] to accomplish the demanded degree of availability and reliability.

While in theory a service is a deterministic state machine, most real-world service implementations are not. This is especially the case for modern services that are implemented using concurrent execution (i. e., threads) to use multiple cores for improving throughput and scalability. In such systems, the execution environment of a service (e. g., the operating system or a managed runtime environment) determines the execution order of threads independently of the service logic. Each request processed by such a service is executed by a dedicated thread. If different requests share data, locks to prevent inconsistencies guard this data. While data is well protected this way, the order in which requests get processed and consequently modify data is not predetermined. Thus, the execution order is likely to differ between multiple instances once replication is applied, ultimately leading to state inconsistencies and divergent client replies, as exemplified by a simple counter example shown in Figure 1.

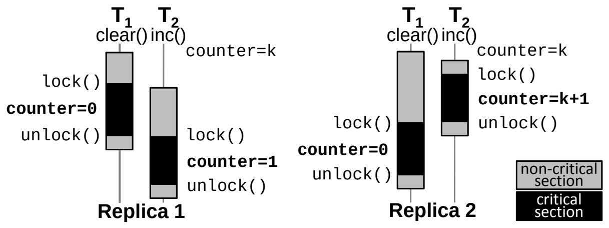

Figure 1: A minimal example for non-determinism caused by concurrency. On replica 1, thread T_1 first clears a lock-protected counter. Then, thread T_2 increments the counter by one. On replica 2, the operations are performed in the opposite order. The final counter state on replica 1 is 1, whereas on replica 2 it is 0.

In sum, state-machine replication and unrestricted concurrent execution of state-sharing threads lead to serious problems. The latter has been recently demonstrated in the course of a software migration testing infrastructure running an old and a new software instance in parallel. It detected a high number of diverging replies that turned out to be false positives that could be blamed on unrestricted concurrency and other causes for non-determinism [7].

A pragmatic approach is to enforce sequential execution by processing only one request at a time [4, 5], which however leads to poor performance and can even cause deadlocks [13]. Therefore, more sophisticated approaches are needed; for example, using some form of lock-step replication [3, 6]. However, this is costly in terms of inter-replica coordination and usually necessitates hardware support. Other approaches settle for restricted forms of parallel execution, but are less demanding in terms of coordination and hardware requirements [1, 2, 10, 12, 13]. While some of them enforce the order of execution of critical sections purely based on the request order as imposed by the replication infrastructure [1, 13], others employ a leader-follower approach in which one leader process communicates the lock order that the other replicas should follow [2, 11, 12]. None of the solutions that handle multithreading comes close to the performance of a non-deterministically–executed variant in a distributed setting, either because of limited parallelism or because of extended communication overhead [8].

Kotla and Dahlin [10] avoid the problem of coordinating shared-data access at the lock level altogether by pre-processing requests via a special module that makes use of application knowledge and restricts parallel execution to non-sharing requests. Domaschka et al. [9] aim at relaxing this restriction by using static code analysis, enabling reachability analysis of locks at runtime based on the current execution path. However, the associated runtime overhead cannot be neglected, and the approach resorts to pessimistic algorithms where static analysis reaches its limits.

In this paper, we present *Storyboard*, an approach that utilizes application knowledge to heuristically predict the execution path of a request. In case of a correct forecast, this enables parallel execution at a similar degree as in an unreplicated scenario. If the predicted execution path turns out to be wrong, a replica establishes a deterministic execution order in cooperation with other replicas. As predictions do not have to be correct, Storyboard can optimistically bet on the most likely outcome in those cases where the execution path heavily depends on the service state or is too complex for a correct prediction. In sum, Storyboard utilizes application knowledge to minimize costly inter-replica coordination and exploit concurrency at a similar level as offered by a non-deterministic execution. This makes it superior to approaches that restrict concurrency in favor of avoiding inter-replica coordination as well as approaches that enable high concurrency but heavily rely on coordination.

Our approach integrates well with a standard replication architecture that consists of an agreement stage and an execution stage. The agreement stage orders the requests and forwards them to a *predictor* component, which in turn makes a forecast on a per-request basis; such a forecast comprises an ordered sequence of locks that are assumed to be taken during request execution. This prediction is based on application knowledge that can be either static or dynamic, but does not need to be totally correct. Before a request is executed, the prediction is handed over to our Storyboard component that is part of the execution stage of each replica; this component monitors the order in which locks are acquired during request execution. If the request advances as predicted, it can be executed straight to completion, and Storyboard enforces that concurrent requests are executed in a deterministic order across all replicas. If a prediction turns out to be wrong, this is detected by Storyboard and resolved by inter-replica coordination, thereby avoiding costly rollback mechanisms.

In the remainder of the paper, we first give a brief overview of the system model. Second, we outline how to predict the execution path of a request and proceed by detailing the subsequent processing. Third, we outline how to handle wrong execution-path predictions. Fourth, we present initial results and finally draw conclusions.

2 System Model

Replicas have to implement a deterministic state machine [14] to ensure consistency. The state machine consists of a set of variables encoding its state and a set of commands operating on them. The execution of a command leads to an arbitrary number of variables being read and/or modified and, as a result of the execution, an output being provided to the environment. For the same sequence of inputs, every non-faulty replica must produce the same sequence of outputs.

To model today's multithreaded high-performance service implementations, we assume a concurrent state machine that is able to execute multiple commands in parallel. Each command is processed in an associated thread. Commands are allowed to access overlapping sets of variables. Such shared variable sets are protected by locks that guarantee that only one command at a time operates on them; other commands that try to access the variable set are blocked temporarily. Depending on the interleaving of threads, there is a set of valid results for a given command. To preserve determinism, the same order has to be imposed across all replicas. For simplicity, we assume that each command is executed by a single thread that is not allowed to spawn further threads, and that the use of non-blocking or wait-free synchronization is not permitted.

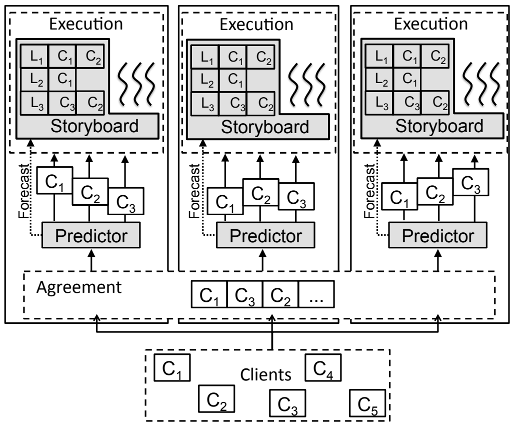

Figure 2: System architecture of Storyboard

3 General Approach

Figure 2 outlines a general replicated state-machine architecture which comprises an agreement stage that enforces an order on the commands issued by one or more clients and an execution stage that represents the service replicas.[1] It is enhanced by Storyboard as follows: In between the agreement stage and the execution stage, there is a *predictor* component that executes a `predict()` function on every ordered command. This function is deterministic and utilizes knowledge about the service and its state. Depending on a given command, it outputs a *forecast*, that is, an ordered sequence of locks that the command presumably acquires to access and modify subsets of the service state during its execution. The forecast is forwarded to our *Storyboard* component via a separate channel. Based on the forecast, Storyboard ensures a deterministic execution order among multiple concurrently executed commands by monitoring and managing their access to locks.

The Storyboard component controls the processing of commands in such a way that two commands sharing state-variable sets never overtake each other regarding the order in which they acquire shared locks. More formally, for every pair of commands C_i and C_j with $i < j$ (i. e., according to the total order imposed by the agreement stage, C_i is executed before C_j), every lock that is shared between C_i and C_j must first be acquired by C_i and then by C_j. Apart from this constraint, an arbitrary number of commands can be executed in parallel, each command at its own speed. In Sections 5 and 6 we discuss some limited relaxations that allow commands to overtake each other in a way that preserves determinism.

In the optimal case, the forecast made by the predictor exactly matches the execution path of a command. In order to always produce a correct forecast, the `predict()` function has to be optimal; for example, for the shared counter presented in Figure 1, an optimal `predict()` function predicts the lock that protects the counter for all commands accessing the counter. This counter is a simple example, but we expect optimal forecasts to be feasible also for much more complex services. This can be achieved, for example, by making a careful code analysis to build the `predict()` function.

However, as the lock sequence of a command may heavily depend on the internal service state, a rather complex logic might be needed to correctly predict the execution. On the other hand, `predict()` should be as simple and fast as possible to compute, as otherwise the benefits of concurrent execution would be degraded. Accordingly, simple heuristic `predict()` functions that might fail to provide a correct forecast for rare corner cases are an attractive alternative; for example, lock traces of common workloads may help to identify the most likely lock sequence for each command type.

In the remainder of the paper we will therefore refrain from a perfect forecast. In fact, Storyboard can handle *mispredictions* that are completely wrong. The Storyboard component detects a misprediction when a command unexpectedly aims to acquire a lock L_{unex} that is not the next lock according to the forecast or when a command finishes prematurely (i. e., the command does not acquire all locks included in the forecast); in the latter case, no further actions besides removing the forecast from the system are required.

In general, upon the detection of a misprediction, the Storyboard component does not immediately grant the lock in question to the command that unexpectedly demands it: as there is no specified lock order, such a procedure could introduce inconsistencies. Instead, Storyboard blocks the execution of the command and instructs the predictor to *repredict* the command's execution path. The predictor reacts by sending a `repredict` command through the agreement stage in order to determine a consistent point in the execution order across all replicas at which the reprediction can be safely performed[2]. Upon receiving the command, a reprediction for the blocked command is performed; this reprediction will at least contain the lock which the command currently seeks to acquire, but possibly also further locks. Finally, Storyboard components on all replicas continue the execution of the command according to the new forecast. In case of further mispredictions, the cycle of execution and reprediction repeats until the command finally completes.

[1]For now, we assume a crash-stop fault model but will discuss how to relax this assumption in Section 7.

[2]Depending on the implementation, there might be either one dedicated predictor component that is responsible for reprediction, typically co-located with the leader of the agreement layer, or all predictor components can initiate repredictions and duplicates have to be suppressed.

4 Normal Operation

For every command, the predictor forwards a forecast to the Storyboard component, which in turn updates a *forecast store*. This store is a list[3] that has a slot for every lock protecting parts of the service state; each slot of the list contains a *FIFO*-ordered list. For every predicted lock, Storyboard inserts a tuple comprising the command identifier and the forecast index into the lock's FIFO list (see Figure 2). Note that updating the forecast store is done sequentially to implement the command order imposed by the agreement stage.

After the forecast store update, the command is executed. At this point, we can relinquish sequential execution, and all commands are processed concurrently at the speed of their associated thread. Thereby, we assume that the thread running on behalf of a command can be identified. Storyboard associates a counter to each command, which is increased every time the thread takes a lock while executing the command.

When a thread wants to enter a critical section, it first has to acquire the associated lock. This procedure is intercepted by the Storyboard component and the forecast store is consulted. In particular, the slot assigned to the lock requested is selected, and it is checked whether the identity of the command matches the first element of the list and whether the counter of the command matches the position in the forecast history. If this is the case, the thread is allowed to enter the critical section. When the thread has finished the critical section, the command identifier is removed from the FIFO list of the lock and the counter is increased by one.

If the command is not in the first position of the FIFO list of the lock, but somewhere else, the thread will be suspended until the command is in the first position. In this way, we enforce the order of execution as determined by the agreement stage. If a lock was not foreseen to be taken by a command, the output of the `predict()` function or the command lock counter do not match the position in the forecast history, the predictor guessed wrong. In this case, we have an out-of-order execution path which requires special actions as outlined next.

5 Handling Mispredictions

There might be cases in which the `predict()` function returns an execution path, but the path taken by a thread will differ, for example, because of an internal service state influencing the execution. In this case, a thread demands to acquire a lock, and likely in the future a sequence of locks, that diverge from what is stored in the forecast store.

[3]For simplicity, we assume a static set of locks, but locks could be added at runtime to the forecast store to match more dynamic scenarios.

We cannot let the thread acquire the lock directly, as this would introduce the same inconsistencies as if we never interfered. Thus, we re-schedule/repredict the request execution path based on the new input in a deterministic way. To do so, we send a `repredict` command via the agreement stage. Once this message has been received, we have a deterministic point in the execution order across all replicas from which we can replan the further processing of the original command.

Independent Locks If a mispredicted lock is independent of all other locks and simply protects some subset of the service state from concurrent access, we can treat the ordered `repredict` command as if it were an entirely new command. Thus, we remove the missed forecast from the forecast store and then feed all the information into the `predict()` function, including the previously requested lock. Depending on the input and the lock requested, `predict()` offers a new forecast that at least includes this lock in the first position.

Nested Locks Handling of nested locks requires further actions. Assume a command C_i tries to acquire a lock L_{unex} not included in its forecast while holding locks. Imagine there is another command C_j that needs a subset of these locks held by the previous request and lock L_{unex} follows in C_j's forecast. Next, the reprediction is performed, and the new forecast for C_i, including L_{unex}, is added to the forecast store. The result is a deadlock. Command C_i cannot proceed with L_{unex} as it was first predicted for C_j, whereas C_j cannot continue execution as it has to wait for locks held by C_i.

We solve this order inversion problem by carefully enqueuing repredicted locks in the forecast store so crossing of nested lock chains of dependent commands is avoided. This is achieved by extending the information stored in the forecast store by a list of nested locks held by a command while acquiring a next lock and a list of locks that will be acquired while holding this lock. In terms of nested locks, we are now able to search in the past and in the future for a certain command at a certain point in its execution path. In case of a misprediction, the reprediction request of a command will be attributed by the list of already acquired locks (past). This information together with the new prediction enable a safe way to insert the command into the forecast store; prior to that, we remove the missed forecast from the forecast store.

For every lock in a prediction, we insert the command information into the list associated to the lock as before. However, in this case, we cannot simply stick with the FIFO order, but have to insert the command at a certain position in the order in which a lock is assumed to be accessed. To find the right spot, we start at the beginning of the list with the logically oldest command request and work consecutively to the end. For each element in the list, we check whether a command shares locks of its prediction (future) with the history of the repredicted command (past). If this is the case, we have to place the repredicted command before the command it is compared with. If the repredicted command does not share locks with any of the commands in the list, it is placed at the end of the list.

6 Condition Variables

Condition variables per se need no special treatment as locks protect them. However, condition variables are used for coordination amongst threads, and this can lead to problems. Imagine that a thread executing on behalf of a command C_i checks some condition that is not fulfilled and waits until the condition is true. The command C_i, however, has a prediction for a set of forthcoming locks. Further, we assume there is a command C_j that at some point in its execution will enable the condition C_i is waiting for and then notify command C_i. In a normal system, this is no problem. However, in the context of Storyboard, C_i might share some forthcoming locks in its prediction with command C_j that have to be acquired by C_j before enabling the condition. In this case, we have a deadlock, because C_j is not allowed to overtake C_i, and C_i waits for notification by C_j.

The problem can be addressed by restricting the prediction of a command to the point where it locks the critical section of the condition variable even if we know more about the command. In this case, after the condition has become true, the command will continue execution and acquire a lock that was not foreseen by the prediction, and consequently a reprediciton is necessary. However, this is only necessary in the special cases outlined. Furthermore, there might be time-bound condition variables where a service only waits for a timeout to expire before execution is continued regardless of whether the condition is fulfilled or not. Because of different execution speeds, there might be replicas in which a condition becomes true before the timeout expires and vice versa. We handle such a non-deterministic timeout by masking it as a reprediction request that is received by all replicas in the same order.

7 Arbitrary Faults

So far, we only considered systems that are subject to crash stop failures. However, Storyboard can be extended to tolerate arbitrary faults using a Byzantine fault-tolerant agreement protocol [4] and some limited extensions. The actual processing of commands does not need to be changed, and the same applies to handling forecasts provided by the `predict()` function. However, extensions are necessary when it comes to mispredictions. In this case, it is not enough to wait for the first indication of a misprediction of a replica, as this might be provided by a malicious node, but until at least $f + 1$ nodes have signaled it, f being the maximum number of faults to tolerate. This ensures at least one correct indication of a misprediction. If this requirement is fulfilled, the workflow outlined for reprediction can be performed. Due to the higher protocol overhead of Byzantine fault-tolerant agreement compared to fail-stop protocols, mispredictions should be avoided; that is, the `predict()` function needs to provide almost optimal results in order to gain benefits from using Storyboard.

8 Preliminary Results

As an initial evaluation, we investigated whether the lock order can be predicted for a realistic network-based service. Therefore, we analyzed the management of shared data in CBASE-FS [10], which implements a network file system (NFS) service. For NFS, command types are limited and every command type can be determined by its header. These are prerequisites for implementing a lean and fast predictor component. We ran some initial benchmarks in which we traced the lock access and the variability of lock order for the various command types.

In the context of the PostMark benchmark, we detected at most three different orders of lock access for the same command type. The number of different locks accessed varied between one and eight, some of them were accessed multiple times. After careful examination of the code, it turned out that the number of locks that have to be considered by the predictor component can be reduced to between one and three, as some locks only protect system-internal buffers that do not directly contribute to the visible service state and therefore can be ignored. This confirms the results gained by Pool et al. for other services [12] and narrows down the number of command types with a variable lock history to only one command type (i. e., lookup). Here, we traced exactly two possible execution paths that are taken depending on whether a queried file exists or not. Taking these facts into account, a predictor can be implemented that is almost optimal for CBASE-FS that builds a conformance wrapper for standard NFS implementations.

We implement Storyboard as a customized POSIX Threads (pthreads) library that can be pre-loaded to any pthreads-based service and are in the process of building the predictor component for CBASE-FS, which then will be integrated with an agreement stage. Thereby, we aim at evaluating Storyboard in the context of fail-stop as well as Byzantine faults by using, for example, Spread and SmartBFT.

9 Conclusion and Future Work

We outlined Storyboard, an approach that supports deterministic multithreaded execution. In contrast to related approaches, it utilizes application-specific knowledge to improve concurrency and avoids inter-replica coordination if possible. Thereby, we chose a predictive approach that does not need to be exact because mispredictions will be resolved at runtime.

Our evaluation results indicate that prediction can have a high success rate for moderate complex services like a remote file system. Accordingly, we are confident that Storyboard will increase throughput of replicated multithreaded services in the context of fail-stop as well as Byzantine fault-tolerance.

10 Acknowledgments

This work was partially supported by the German Research Council (DFG) under grant no. KA 3171/1 and by the MASTER research project funded by the European Commission's FP7 programme.

References

[1] BASILE, C., KALBARCZYK, Z., AND IYER, R. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the 2003 Int. Conf. on Dependable Systems and Networks (DSN '03)* (2003), pp. 149–158.

[2] BASILE, C., WHISNANT, K., KALBARCZYK, Z., AND IYER, R. Loose synchronization of multithreaded replicas. In *Proceedings of the 21st IEEE Symp. on Reliable Distributed Systems (SRDS '02)* (2002), pp. 250–255.

[3] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS) 14*, 1 (1996), 80–107.

[4] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symp. on Operating Systems Design and Implementation (OSDI '99)* (1999), pp. 173–186.

[5] CLEMENT, A., KAPRITSOS, M., LEE, S., WANG, Y., ALVISI, L., DAHLIN, M., AND RICHE, T. UpRight cluster services. In *Proceedings of the 22nd Symp. on Operating Systems Principles (SOSP '09)* (2009), pp. 277–290.

[6] CULLY, B., LEFEBVRE, G., MEYER, D., FEELEY, M., HUTCHINSON, N., AND WARFIELD, A. Remus: high availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symp. on Networked Systems Design and Implementation (NSDI '08)* (2008), pp. 161–174.

[7] DING, X., HUANG, H., RUAN, Y., SHAIKH, A., PETERSON, B., AND ZHANG, X. Splitter: a proxy-based approach for post-migration testing of web applications. In *Proceedings of the 5th European Conf. on Computer Systems (EuroSys '10)* (2010), pp. 97–110.

[8] DOMASCHKA, J., BESTFLEISCH, T., HAUCK, F. J., REISER, H. P., AND KAPITZA, R. Multithreading strategies for replicated objects. In *Proceedings of the ACM/IFIP/USENIX 9th Int. Middleware Conf. (Middleware '08)* (2008), pp. 104–123.

[9] DOMASCHKA, J., SCHMIED, A. I., REISER, H. P., AND HAUCK, F. J. Revisiting deterministic multithreading strategies. In *Proceedings of the IEEE Int. Parallel and Distributed Processing Symp. (IPDPS '07)* (2007), pp. 225–232.

[10] KOTLA, R., AND DAHLIN, M. High throughput Byzantine fault tolerance. In *Proceedings of 2004 Int. Conf. on Dependable Systems and Networks (DSN '04)* (2004), pp. 575–584.

[11] NAPPER, J., ALVISI, L., AND VIN, H. A fault-tolerant Java virtual machine. In *Proceedings of the 2003 Int. Conf. on Dependable Systems and Networks (DSN '03)* (2003), pp. 425–434.

[12] POOL, J., WONG, I. S. K., AND LIE, D. Relaxed determinism: making redundant execution on multiprocessors practical. In *Proceedings of the 11th USENIX Work. on Hot Topics in Operating Systems (HOTOS '07)* (2007), pp. 1–6.

[13] REISER, H. P., HAUCK, F. J., DOMASCHKA, J., KAPITZA, R., AND SCHRÖDER-PREIKSCHAT, W. Consistent replication of multithreaded distributed objects. In *Proceedings of the 25th IEEE Symp. on Reliable Distributed Systems (SRDS'06)* (2006), pp. 257–266.

[14] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Survey 22*, 4 (1990), 299–319.