UNIVERSIDADE DE LISBOA
Faculdade de Ciências
Departamento de Informática

# DEPENDABLE DATA STORAGE WITH STATE MACHINE REPLICATION

**Marcel Henrique dos Santos**

**DISSERTAÇÃO**

MESTRADO EM INFORMÁTICA

2014

UNIVERSIDADE DE LISBOA

Faculdade de Ciências

Departamento de Informática

# DEPENDABLE DATA STORAGE WITH STATE MACHINE REPLICATION

**Marcel Henrique dos Santos**

**DISSERTAÇÃO**

MESTRADO EM INFORMÁTICA

Dissertação orientada pelo Professor Doutor Alysson Neves Bessani

2014

# Acknowledgements

I would like to thank my advisor and friend, Professor Alysson for all the help and encouragement he provided before and during the Master's program. I certainly would not get this far without his advices and knowledge. I would like also to thank his wife and my friend Cassia for the support while we were in Portugal.

I thank also my wife Fernanda for being there for me during this whole journey. She did not think twice when I asked her to move with me to Portugal for two years to work on the TClouds project. She also listened to me during all good and bad times I had during these two years. My mother in law Neusa, thank you for being such a great mother for Fernanda and also a friend to myself.

My colleagues from FCUL helped a lot during this time, with answers, ideas and even digging with the code for problems I had. Thank you all for that. A special thanks to Vinicus Cogo, without whom I would not be able to deliver this dissertation. Another thanks to Pedro Gonçalves who helped me with all the invoices, expenses reports, contracts and paperwork I needed in these two years. His dedication to his job is certainly something that Lasige could not work without.

Finally I would like to thank all my friends and family for the support they gave me during these years. Thank you Maycol and Cris for the review. My parents Laerte and Wilma, thank you for the education and values you gave me, which made me get here. My brother Vladimir and sister Evelise, thank you for being my friends and helping me with all I needed while I was not in Brazil.

This work was supported by the EC FP7 through the project TClouds (ICT-257243) and by FCT through project LaSIGE (PEst-OE/EEI/UI0408/2014).

*To my father Laerte, for buying the IBM XT, where it all started.*

# Abstract

State Machine Replication (SMR) is a technique to replicate information across servers, also called replicas, providing fault tolerance to services. Instead of execute in a single server, requests from multiple clients are ordered and executed in a set of replicas. Results are confirmed to the clients once a predefined quorum of replicas replies. Several studies prove possible to tolerate up to $f$ faults using $2f + 1$ replicas. Byzantine Fault Tolerant (BFT) SMR configurations, where replicas can behave in an arbitrary mode, require $f$ additional replicas, with the total of $3f + 1$ replicas.

When a replica is detected faulty, it has to be recovered with an updated state to reduce the vulnerability of the system. This state is generated during the service execution, when write operations are logged. To bind the size of the log and the time to replay it, periodic snapshots of the service state, or checkpoints, are taken and the log reset. During recovery the checkpoint and the log are transferred from other replicas.

To provide resilience to co-related faults, information has to be logged in durable storage. Synchronous writes in durable storage and constant checkpoints can affect throughput and latency of the system as replicas have to wait for information to be stored before reply. When a checkpoint is being taken the system cannot make progress because the state cannot be changed. This may cause the service to be interrupted for several seconds during a checkpoint. The state transfer to a recovering replica can also cause perturbations in the system execution, as correct replicas has to read and transfer the state, composed by the checkpoint, log and digests of messages in case of BFT systems.

In this dissertation we present three techniques to improve the performance of state storage and transfer in a BFT SMR protocol - BFT-SMART. The first, *Parallel Logging* stores information in the log in parallel with its execution by the application. The second, *Sequential Checkpointing* makes only one replica take a checkpoint at a time, in a round-robin fashion, allowing the system to make progress during that period. The last technique, *Collaborative State Transfer (CST)* reduces the perturbation in a system during state transfer to a recovering replica by having one replica providing the checkpoint and the remaining providing portions of the log. We also present algorithms that address the problem of co-related failures. When several replicas fail at the same time it is possible to start them simultaneously and compare the stored state before having the service available again. After presenting the techniques, we provide a prototype implementation called Dura-SMaRt with an evaluation against BFT-SMART to compare the efficiency of the new techniques. We performed the evaluation with two applications: a consistent key-value store – SCKV-store – and a coordination service that stores information in tuple spaces – DepSpace.

Next, we evaluate Dura-SMaRt in a complex use, having a database replication middleware built on top of it. SteelDB, provide fault tolerance for transaction processing in database management systems (DBMS).

Transactional databases provide durability for information systems executing operations inside boundaries called transactions. Transactions guarantee several properties, amongst which, atomicity and isolation. Atomicity enforces that all operations executed inside a transaction are confirmed, or none is. Isolation guarantees that operations inside a transaction are only visible for other transactions after it is finished. Concurrency mechanisms implementations allow several transactions, from several clients to be executed at the same time, improving the performance of a DBMS. To provide dependability to DBMS, several DBMS vendors provide replications mechanisms that usually rely on the efficiency of fail detection and recovery. Such replication mechanisms are also attached to the vendor implementation. With SteelDB we provide transparent Byzantine fault tolerance with $3f + 1$ replicated databases. SteelDB requires no changes in the client code as it provides a driver implementation of the JDBC specification. Clients have only to switch the current driver provided by the database vendor it is using to the driver provided by SteelDB.

After describing the concepts and implementation of SteelDB we present an evaluation performed on SteelDB during the last year of the FP7 TClouds project. We evaluated SteelDB for functional and performance aspects with a real application executing different types of transactions and comparing results with executions on different environments. We compared SteelDB executions in local area networks, private, public and hybrid clouds discussing the differences in performance and efficiency of optimizations present in the middleware.

After SteelDB evaluation we discuss the related work to state management in SMR and database replication middlewares.

Finally we will conclude the work with a discussion on the results obtained and purposes for future work.

**Keywords:** Dependability, Replication, Fault Tolerance, Database, Disaster Recovery

# Resumo

Replicação de Máquina de Estados (SMR) é uma técnica para replicar informações entre vários servidores, também chamados de réplicas, provendo tolerância a faltas para aplicações. Ao invés de executar os pedidos dos clientes em um único servidor, pedidos de vários clientes que alteram o estado de uma aplicação passam por um protocolo de ordenação e são entregues na mesma ordem para um conjunto de réplicas. Os resultados somente são confirmados aos clientes após um quórum pré-definido de réplicas responder. Vários estudos provaram ser possível tolerar até $f$ faltas com o uso de $2f + 1$ réplicas. Configurações para SMR com Tolerância a Faltas Bizantinas (BFT), onde réplicas podem apresentar comportamento arbitrário, necessitam de $f$ réplicas adicionais, com o total de $3f + 1$ réplicas.

Quando uma réplica percebe que esta atrasada em relação às demais, ou uma nova réplica é adicionada ao sistema, ela precisa instalar uma a versão atualizada do estado, para poder participar do protocolo de ordenação e processamento dos pedidos, restaurando assim a tolerância do sistema a faltas. Réplicas geram um *log* das operações executadas para terem uma cópia atualizada do estado, necessária a uma possível recuperação. As operações de escrita são armazenadas de forma sequencial no *log*. Para limitar seu tamanho e o tempo para reproduzí-lo em uma réplica que está recuperar-se, as réplicas tiram cópias do estado periodicamente em *checkpoints* e, apagam o *log* em seguida. Durante a recuperação de uma réplica, o *checkpoint* e o *log* são transferidos pelas demais. A réplica que está a recuperar-se instala o checkpoint recebido e executa as operações do *log* antes de confirmar às demais que está pronta a processar pedidos novamente.

Para oferecer tolerância a faltas co-relacionadas, onde várias réplicas podem apresentar falhas ao mesmo tempo, informações precisam ser armazenadas em mídia durável. Escritas síncronas em mídia durável e *checkpoints* constantes podem diminuir o *throughput* e aumentar a latência do sistema pois as réplicas precisam esperar até que a escrita seja concluída, antes de confirmar a operação ao cliente. De outra forma, no caso de uma falha antes do fim da escrita, poderíamos ter dados confirmados ao cliente mas não armazenados. Realizamos experimentos que provam que a substituição da mídia por opçoes mais rápidas, nomeadamente, disco rígido por SSD, apesar de diminuir o tempo de escrita ainda afeta consideravelmente o *throughput* da aplicação.

Enquanto um *checkpoint* do estado é gerado, a aplicação não pode estar a processar operações de escrita, pois estas podem alterar este estado. Isto faz com que o throughput do sistema seja zero durante este período, que pode demorar vários segundos, dependendo do tamanho do estado. Conforme demonstramos através de gráficos de desempenho da aplicação, a transferência de estado a uma réplica que está a recuperar-se pode também causar perturbações nas réplicas que estão a transferí-lo, pois estas precisam ler dados em mídia durável e transferir o estado pela rede. Em situações onde o tamanho do estado

é grande, a tranferência pode afectar a comunicação com as demais réplicas e com os clientes.

Apresentamos neste trabalho três técnicas puramente algorítmicas que melhoram o desempenho no armazenamento e transferência de estado em um protocolo BFT SMR chamado BFT-SMaRt. A primeira, *Parallel Logging*, faz as réplcias armazenarem as operações no log em paralelo com sua execução pela aplicação. Em aplicações onde o tempo para se executar uma operação é considerável, pode-se reduzir o tempo total ao executar a operação e o log em *threads* diferentes. A segunda, *Sequential Checkpointing* faz somente uma das réplicas tirar um *checkpoint* por vez, sequencialmente, permitindo ao sistema fazer progresso nesse período. A terceira técnica, *Collaborative State Transfer (CST)* define uma estratégia para transferência de estado onde uma réplica envia o *checkpoint* da aplicação e as demais enviam partes do *log*, reduzindo o efeito da transferência de estado nas réplicas que estão a enviá-lo. Apresentamos também algorítmos para resolver o problema de faltas co-relacionadas. No caso de uma falta onde todas as réplicas vão abaixo, é possível fazê-las retomar o serviço e iniciar a execução novamente, após iniciadas.

Implementamos as novas técnicas apresentadas em um protótipo chamado Dura-SMaRt para obtermos uma avaliação de seu efeito no desempenho de um sistema replicado. Apresentamos uma avaliação do protótipo e do BFT-SMaRt com duas aplicações diferentes construídas sobre estes, uma *consistent key-value store* chamada SCKV-Store e um serviço de coordenação que utiliza um espaço de tuplos para armazenamento de dados chamado DepSpace.

Comparamos os resultados de diversos experimentos para demonstrar que as novas técnicas reduzem o impacto da escrita de operações em mídia durável. Apresentamos resultados que mostram que a execução das operações de escrita em paralelo com seu armazenamento no log não afectam o throughput em para aplicaçoes onde o tempo de execuçao de mensagens é considerável. As novas técnicas também reduzem o impacto que a geração de um checkpoint tem no *throughput* do sistema. Por fim demonstramos que a transferência de estado tem menor impacto no *throughput* do sistema com as novas técnicas quando comparadas ao modelo anterior onde uma réplica era responsável por enviar o *checkpoint* e o *log* das operações.

De seguida, avaliamos o Dura-SMaRt em um caso de uso complexo: um *middleware* para replicação de bases de dados chamado SteelDB. Este *middleware* utilizou o Dura-SMaRt para replicação de dados, oferecendo tolerância a faltas para transações em sistemas de gerenciamento de bases de dados (DBMS).

Bases de dados transacionais fornecem durabilidade para sistemas de informação ao executar operações dentro de barreiras chamadas transações. Uma transação garante algumas propriedades, entre as quais atomicidade e isolamento. Atomicidade implica que todas as operações executadas são confirmadas, ou nenhuma é. Isolamento garante que

alterações presentes dentro de uma transação só serão visíveis às demais após o fim desta. Estas propriedades permitem a utilização da base de dados simultaneamente por vários clientes, aumentando a concorrência na execução de operações. Para aumentar a disponibilidade e recuperação a faltas, vários desenvolvedores de DBMS fornecem mecanismos de replicação de dados. Estes mecanismos geralmente estão ligados a eficiência dos sistemas de detecção de falha e recuperação. Eles também estão intrinsicamente ligados ao fabricante da base de dados escolhido. Com o SteelDB nós oferecemos tolerância transparente a faltas Byzantinas, com o uso de $3f + 1$ bases de dados. O SteelDB fornece aos clientes uma implementação da especificação JDBC, portanto, clientes que já utilizam um *driver* JDBC para aceder a uma base de dados, somente precisam trocá-lo pelo *driver* fornecido pelo SteelDB.

Depois de descrever os conceitos e implementação do *middleware* SteelDB, apresentamos uma avaliação deste, realizada no último ano do projeto FP7 TClouds. Esta avaliação testou diversos aspectos de desempenho e funcionalidade em uma aplicação real com diversos tipos de transações, fornecida por um dos parceiros do projeto. Descrevemos a configuração e execução do SteelDB em diversos ambientes como redes locais, *clouds* privadas, públicas e híbridas. Comparamos de seguida os resultados da execução nestes diferentes ambientes para avaliar a eficiência de optimizações incluídas no *middleware*. Apesar da utilização de bases locais ter desempenho consideravelmente melhor com relaçao à replicação com o SteelDB, bases locais não fornecem tolerância a faltas. Também demonstramos que quando o tamanho das transações aumenta, a diferença entre os tempos de execução diminui, evidenciando o custo da troca de mensagens entre redes remotas. Otimizações incluídas no SteelDB, entretanto, diminuem o número de mensagens necessarias por operação, reduzindo também o seu tempo de execução total.

Avaliamos também o desempenho do SteelDB em simulações com diferentes tipos de faltas. Nos casos de teste que avaliamos, as faltas não afectam consideravelmente o desempenho do SteelDB, uma vez que o protocolo de replicação Dura-SMaRt não precisa esperar por respostas de todas as réplicas antes de confirmar as operações aos clientes.

Após apresentarmos a avaliação do SteelDB, discutimos os trabalhos relacionados com o gerenciamento de estado em sistemas SMR e também estudos e alternativas para replicação de bases de dados com o uso de SMR.

Concluímos com uma discussão dos resulados obtidos e propostas de trabalhos futuros.

**Palavras-chave:** Segurança de Funcionamento, Replicação, Tolerância a Faltas, Bases de Dados, Recuperação de Desastres

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Context

Information systems with client-server architectures are a common approach to manipulate data since decades ago.

Although deployments with a single server can tolerate multiple requests at the same time, they can represent a single point of failure. Problems like server crashes, network disruption, software bugs or malicious faults can lead to the unavailability of the service or even data corruption.

Replication is an alternative to provide resilience and availability for information systems. It allows the system to tolerate faults on a predefined number of servers, according to the defined fault model.

### 1.1.1 State Machine Replication

State Machine Replication (SMR) [54] is a technique to replicate data through different servers (replicas), providing tolerance to faults in up to a predefined number of replicas.

The SMR model requires replicas to be in the same state after executing the same set of requests, to make possible a consistent global state [54]. To provide that, all replicas need to start in the same state $S_0$ and execute operations that change the state in the same order. An additional requirement is that operations must be deterministic.

To make possible for all replicas to execute update operations in the same order, a total order multicast [30] protocol can be used. This protocol enforces that all messages are delivered in the same order for all participants. If, for instance, one participant receives a sequence of messages having a message from client *A* after a message from client *B*, then all participants will receive a sequence of messages having client *B* after client *A*.

Many systems in production use variations of this approach to tolerate *crash faults* (e.g., [11, 12, 16, 20, 32]). Usually, in a crash fault tolerant system (CFT), the number of replicas required to tolerate $f$ *crash faults* is $2f + 1$. Some works shows that it is possible to reduce this number using some assumptions. Research systems have also shown that

SMR can be employed to tolerate *Byzantine faults* [43] with reasonable costs. Usually Byzantine fault tolerant (BFT) systems require $3f + 1$ replicas to tolerate $f$ faults (e.g., [13, 17, 29, 36, 40, 59, 58]). A system with Byzantine faults presents arbitrary behavior, not only crashing or delaying messages but also corrupting its local state or presenting incorrect or inconsistent output.

### 1.1.2 Durability in State Machine Replication

The state transfer protocol is a protocol where correct replicas in a SMR system provide an updated version of the application state to a new or recovering replica. After receiving the current state, the replica can update itself and be able to participate in the system, restoring or increasing the fault tolerance threshold.

A common approach for replicas to manage state is to store the sequence of update messages in a log, transferring the log to a replica when it requests the current state. To bound the size of the log and reduce the number or operations to be executed during a recover, checkpoints of the application state can be taken. When a replica requests the current state, the others transfer the last checkpoint taken plus the log of messages to be replayed.

### 1.1.3 BFT-SMaRt

BFT-SMaRt [8] is a Byzantine Fault Tolerant State Machine Replication library that started to be developed in 2007 to implement a BFT total order multicast protocol for the replication layer of the DepSpace coordination service [9]. In 2009 the development was ramped to implement a complete BFT replication library.

BFT-SMaRt was developed using the Java language and offers a small and clear API to build clients and services on top of it. BFT-SMaRt was designed from the beginning with some principles intended to provide a robust yet efficient BFT SMR library. To ease the utilization from clients, it provides a simple and extensible API with methods for clients to invoke operations on a service. On the server side the library provide several classes that can be extended to perform different operations like execution of ordered and unordered requests, batch of ordered requests and state management.

## 1.2 Motivation

Durable state management is commonly overlooked in several SMR studies. State size may be too small to be considered or studies focuses on performance of fault free executions disregarding state transfer protocols.

The integration of durability techniques – logging, checkpointing, and state transfer – with the SMR approach can be difficult [16]. First of all, these techniques can drastically decrease the performance of a service[1]. In particular, *synchronous logging* can make the system throughput as low as the number of appends that can be performed on the disk per second, typically just a few hundreds [39]. Although the use of SSDs can alleviate the problem, it cannot solve it completely (see Section 2.2). Additionally, *checkpointing* requires stopping the service during this operation [13, 16], unless non-trivial optimizations are used at the application layer, such as copy-on-write [16, 17]. Moreover, recovering faulty replicas involves running a *state transfer protocol*, which can impact normal execution, as correct replicas need to transmit their state.

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

SMR implementations usually uses as applications key-value stores or even simpler test cases like counters. While such applications may be useful to evaluate aspects like latency and throughput, in practice they do not require complex or large states to be taken and stored by the service. Clients with concurrent executions of transactions and multiple sessions constitute a complex exercise to attest the functionality of the state management protocol.

## 1.3 Goals

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

The techniques are three: *parallel logging*, for diluting the latency of synchronous logging; *sequential checkpointing*, to avoid stopping the replicated system during check-

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

points; and *collaborative state transfer*, for reducing the effect of replica recoveries on the system performance. This is the first time that the durability of fault-tolerant SMR is tackled in a *principled* way with a set of algorithms organized in an *abstraction* to be used between SMR protocols and the application.

The proposed techniques were implemented in a durability layer on the BFT-SMaRt state machine replication library [8], on top of which we built two typical SMR-based services: a consistent key-value store (SCKV-Store) and a non-trivial BFT coordination service (Durable DepSpace). Our experimental evaluation shows that the proposed techniques can remove most of the performance degradation due to the addition of durability.

Furthermore, we implemented a complex system in which our novel techniques were employed: SteelDB, a database replication middeware allows replication of database management systems requiring no changes in the client or server codebase. SteelDB efficiently manages state transfer through database replicas requiring no changes in the BFT-SMaRt library. It also does not require knowledge of database internals, using only common database dumps to manage snapshots of the state plus information on open sessions to keep open connections.

## 1.4 Contributions

This dissertation presents the following contributions:

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

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

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

4. The description of an implementation these techniques in BFT-SMaRt, and an experimental evaluation under write-intensive loads, highlighting the performance limitations of previous solutions and how our techniques mitigate them;

5. The implementation of database replication middleware on top of BFT-SMaRt, with efficient and durable state management. After that we present an experimental evaluation of the middleware with replicas in public and private clouds, providing fault tolerance to typical database systems including disaster recovery.

   Portions of this work were published both in conferences and project deliverables [4, 7, 46].

## 1.5 Document Organization

This document is organized as follows:

Chapter 2 discusses the problems encountered by dealing with the state management in state machine replication protocols. It demonstrates that taking checkpoints and logs and dealing with state transfer can affect the performance of a system. It also proposes three new techniques to alleviate the cost of state management.

Chapter 3 describes the implementation of the new techniques over BFT-SMaRt in a prototype we called Dura-SMaRt. We present the architecture created for the durability layer included and a brief description of the services implemented. After that we evaluate the new techniques showing how they can improve the performance in state management while providing durability to SMR.

Chapter 4 describes the effort made to create a complex service (SteelDB) on top of the durable BFT-SMaRt using the techniques described and providing durability for database management systems.

Chapter 5 describes the evaluation of SteelDB. It describes the environment and results achieved replicating a database management system over public and private clouds.

Chapter 6 discusses the related work performed in the area of state machine replication and state management. More specifically, it discusses how previous works dealt with state management and how our work can improves what was done. We also discuss previous works on database replication middleware and common issues faced during the design and implementation of such systems.

Chapter 7 concludes this work with an overview of the work that was done and the issues we had to deal during the execution. It will also point out possible ideas to be explored in the future.

# Chapter 2

# Improving the Efficiency of Durable State Machine Replication

This chapter presents the durable SMR model, and then analyzes the effect of durability mechanisms on the performance of the system. We start by discussing the several performance problems that the creation of checkpoints and logs can cause. Also we will discuss the effect of a state transfer in the execution of operations by replicas. After that we will propose algorithms to alleviate the cost of state management while providing durability to the system.

## 2.1 System Model and Properties

We follow the standard SMR model [54]. Clients send requests to invoke operations on a service, which is implemented in a set of replicas (see Figure 2.1). Operations are executed in the same order by all replicas, by running some form of agreement protocol. Service operations are assumed to be deterministic, so an operation that updates the state (abstracted as a *write*) produces the same new state in all replicas. The state required for processing the operations is kept in main memory, just like in most practical applications for SMR [11, 16, 32].

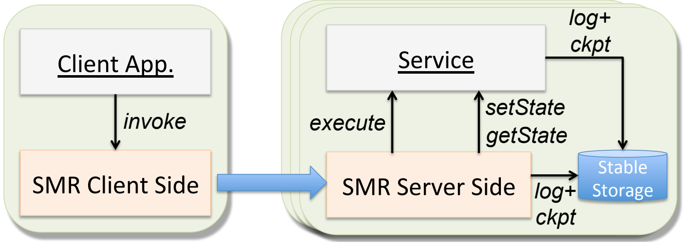


Figure 2.1: A durable state machine replication architecture.

The replication library implementing SMR has a client and a server side (layers at the bottom of the figure), which interact respectively with the client application and the service code. The library ensures standard safety and liveness properties [13, 42], such as correct clients eventually receive a response to their requests if enough synchrony exists in the system.

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

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

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

Traditional mechanisms for enforcing durability in SMR-based main memory databases are logging, checkpointing and state transfer [16, 26]. A replica can recover from a crash by using the information saved in stable storage and the state available in other replicas. It is important to notice that a recovering replica is considered faulty *until it obtains enough data to reconstruct the state* (which typically occurs after state transfer finishes).

*Logging* writes to stable storage information about the progress of the agreement protocol (e.g., when certain messages arrive in Paxos-like protocols [16, 35]) and about the operations executed on the service. Therefore, data is logged either by the replication library or the service itself, and a record describing the operation has to be stored before a reply is returned to the client.

The replication library and the service code synchronize the creation of checkpoints with the truncation of logs. The service is responsible for generating snapshots of its state (method *getState*) and for setting the state to a snapshot provided by the replication library (method *setState*). The replication library also implements a *state transfer* protocol to initiate replicas from an updated state (e.g., when recovering from a failure or if they are too late processing requests), akin to previous SMR works [13, 14, 16, 17, 52]. The state is fetched from the other replicas that are currently running.

## 2.2 Identifying Performance Problems

This section discusses performance problems caused by the use of logging, checkpointing and state transfer in SMR systems. We illustrate the problems with a consistent key-value store (SCKV-Store) implemented using BFT-SMaRt [8]. In any case, the results in the chapter are mostly orthogonal to the fault model and also affect systems subject to only crash faults. We consider write-only workloads of 8-byte keys and 4kB values, in a key space of 250k keys, which creates a service state size of 1GB in 4 replicas. A complete description of our experimental environment appears in Section 3.3.2.

### 2.2.1 High Latency of Logging

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

| Metric | No log | Asynchronous | Synchronous SSD | Synchronous Disk |
|---|---|---|---|---|
| Minimun Latency (ms) | 1.98 | 2.16 | 2.89 | 19.61 |
| Peak Throughput (ops/s) | 4772 | 4312 | 1017 | 63 |

Table 2.1: Effect of logging on the SCKV-Store. Single-client minimum latency and peak throughput of 4kB-writes.

The table shows that *synchronous*[1] logging to disk can cripple the performance of such system. To address this issue, some works have suggested the use of faster non-volatile memory, such as flash memory solid state drives (SSDs) or/in NVCaches [52]. As the table demonstrates, there is a huge performance improvement when the log is written synchronously to SSD storage, but still only $23\%$ of the "No log" throughput is achieved. Additionally, by employing specialized hardware, one arguably increases the costs and the management complexity of the nodes, especially in virtualized/cloud environments where such hardware may not be available in all machines.

There are works that avoid this penalty by using *asynchronous* writes to disk, allowing replicas to present a performance closer to the main-memory system (e.g., Harp [44] and BFS [13]). The problem with this solution is that writing asynchronously does not give durability guarantees if all the replicas crash (and later recover), something that production systems need to address as correlated failures do happen [21, 24, 47, 53].

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

### 2.2.2 Perturbations Caused by Checkpoints

Checkpoints are necessary to limit the log size, but their creation usually degrades the performance of the service. Figure 2.2 shows how the throughput of the SCKV-Store is affected by creating checkpoints at every 200k client requests. Taking a snapshot after processing a certain number of operations, as proposed in most works in SMR (e.g., [13,

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

42]), can make the system halt for a few seconds. This happens because requests are no longer processed while replicas save their state. Moreover, if the replicas are not fully synchronized, delays may also occur because the necessary agreement quorum might not be available.

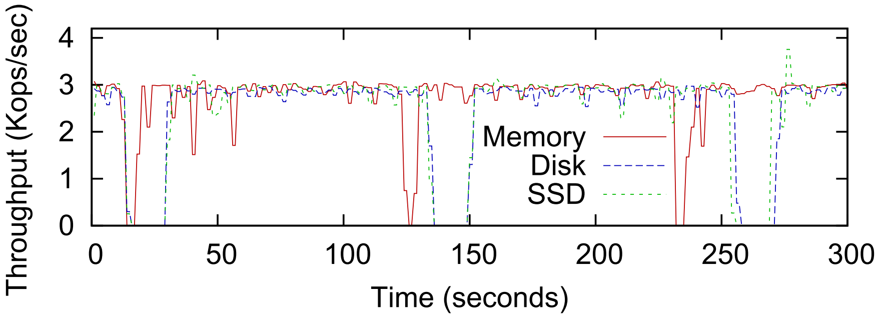


Figure 2.2: Throughput of a SCKV-Store with checkpoints in memory, disk and SSD considering a state of 1GB.

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

Often, the performance decrease caused by checkpointing is not observed in the literature, either because no checkpoints were taken or because the service had a very small state (e.g., a counter with 8 bytes) [13, 18, 29, 36, 40, 59, 58]. Most of these works were focusing on ordering requests efficiently, and therefore checkpointing could be disregarded as an orthogonal issue. Additionally, one could think that checkpoints need only to be created sporadically, and therefore, their impact is small on the overall execution. We argue that this is not true in many scenarios. For example, the SCKV-Store can process around 4700 4kB-writes per second, which means that the log can grow at the rate of more than 1.1 GB/min, and thus checkpoints need to be taken rather frequently to avoid outrageous log sizes. Leader-based protocols, such as those based on Paxos, have to log information about most of the exchanged messages, contributing to the log growth. Furthermore, recent SMR protocols require frequent checkpoints (every few hundred operations) to allow the service to recover efficiently from failed speculative request ordering attempts [29, 36, 40].

Some systems use *copy-on-write* techniques for doing checkpointing without stopping replicas (e.g., [17]), but this approach has two limitations. First, copy-on-write may be complicated to implement at application level in non-trivial services, as the service needs to keep track of which data objects were modified by the requests. Second, even if such techniques are employed, the creation of checkpoints still consumes resources and degrades the performance of the system. For example, writing a checkpoint to disk makes logging much slower since the disk head has to move between the log and checkpoint files, with the consequent disk seek times. In practice, this limitation could be addressed in part with extra hardware, such as by using two disks per server.

Another technique to deal with the problem is *fuzzy snapshots*, used in ZooKeeper [32]. A fuzzy snapshot is essentially a checkpoint that is done without stopping the execution of operations. The downside is that some operations may be executed more than once during recovery, an issue that ZooKeeper solves by forcing all operations to be idempotent. However, making operations idempotent requires non-trivial request pre-processing before they are ordered, and increases the difficulty of decoupling the replication library from the service [32, 35].

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

### 2.2.3 Perturbations Caused by State Transfer

When a replica recovers, it needs to obtain an updated state to catch up with the other replicas. This state is usually composed of the last checkpoint plus the log up to some request defined by the recovering replica. Typically, (at least) another replica has to spend resources to send (part of) the state. If checkpoints and logs are stored in a disk, delays occur due to the transmission of the state through the network but also because of the disk accesses. Delta-checkpoint techniques based, for instance, on Merkle trees [13] can alleviate this problem, but cannot solve it completely since logs have always to be transferred. Moreover, implementing this kind of technique can add more complexity to the service code.

Similarly to what is observed with checkpointing, there can be the temptation to disregard the state transfer impact on performance because it is perceived to occur rarely. However, techniques such as replica rejuvenation [31] and proactive recovery [13, 56] use state transfer to bring refreshed replicas up to date. Moreover, reconfigurations [45] and even leader change protocols (that need to be executed periodically for resilient BFT replication [18]) may require replicas to synchronize themselves [13, 55]. In conclusion, state transfer protocols may be invoked much more often than when there is a crash and a subsequent recovery.

Figure 2.3 illustrates the effect of state transmission during a replica recovery in a 4 -node BFT system using the PBFT's state transfer protocol [13]. This protocol requires just one replica to send the state (checkpoint plus log) – similarly to crash FT Paxos-based systems – while others just provide authenticated hashes for state validation (as the sender of the state may suffer a Byzantine fault). The figure shows that the system performance drops to less than 1/3 of its normal performance during the 30 seconds required to complete state transfer. While one replica is recovering, another one is slowed because it is

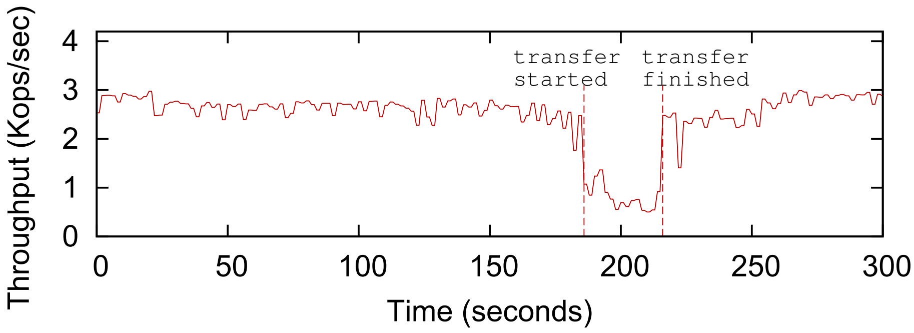


Figure 2.3: Throughput of a SCKV-Store when a failed replica recovers and asks for a state transfer.

sending the state, and thus the remaining two are unable to order and execute requests (with $f = 1$, quorums of $3$ replicas are needed to order requests).

One way to avoid this performance degradation is to ignore the state transfer requests until the load is low enough to process both the state transfers and normal request ordering [32]. However, this approach tends to delay the recovery of faulty replicas and makes the system vulnerable to extended unavailability periods (if more faults occur). Another possible solution is to add extra replicas to avoid interruptions on the service during recovery [56]. This solution is undesirable as it can increase the costs of deploying the system.

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

## 2.3 Efficient Durability for SMR

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

### 2.3.1 Parallel Logging

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

The first idea explores the fact that disks have a high bandwidth, so the latency for writing 1 or 100 log entries can be similar, but the throughput would be naturally increased by a factor of roughly 100 in the second case. This technique requires the replication library to deliver groups of service operations (accumulated during the previous batch execution) to allow the whole batch to be logged at once, whereas previous solutions normally only provide single operations, one by one. Notice that this approach is different from the batching commonly used in SMR [13, 18, 40], where a group of operations is ordered together to amortize the costs of the agreement protocol (although many times these costs include logging a batch of requests to stable storage [42]). Here the aim is to pass batches of operations from the replication library to the service, and a batch may include (batches of) requests ordered in *different agreements*.

The second idea requires that the requests of a batch are processed while the corresponding log entries are being written to the secondary storage. Notice, however, that a reply can only be sent to the client after the corresponding request is executed *and logged*, ensuring that the result seen by the client will persist even if all replicas fail and later recover. Naturally, the effectiveness of this technique depends on the relation between the time for processing a batch and the time for logging it. More specifically, the interval $T_k$ taken by a service to process a batch of $k$ requests is given by $T_k = max(E_k, L_k)$, where $E_k$ and $L_k$ represent the latency of executing and logging the batch of $k$ operations, respectively. This equation shows that the most expensive of the two operations (execution or logging) defines the delay for processing the batch. For example, in the case of the SCKV-Store, $E_k \ll L_k$ for any $k$, since inserting data in a hash table with chaining (an $\mathcal{O}(1)$ operation) is much faster than logging a 4kB-write (with or without batching). This is not the case for Durable DepSpace, which takes a much higher benefit from this technique, as will be demonstrated in Section 3.3.3.

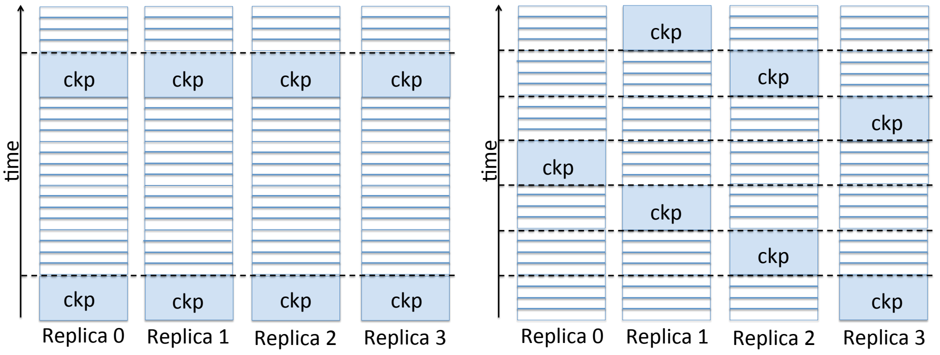
### 2.3.2 Sequential Checkpointing

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

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

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

Figure 2.4 compares a synchronous (or coordinated) checkpoint with our technique. Time grows from the bottom of the figure to the top. The shorter rectangles represent the logging of an operation, whereas the taller rectangles correspond to the creation of a checkpoint. It can be observed that synchronized checkpoints occur less frequently



(a) Synchronized.

(b) Sequential.

Figure 2.4: Checkpointing strategies (4 replicas).

than sequential checkpoints, but they stop the system during their execution whereas for sequential checkpointing there is always an agreement quorum of 3 replicas available for continuing processing requests.

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

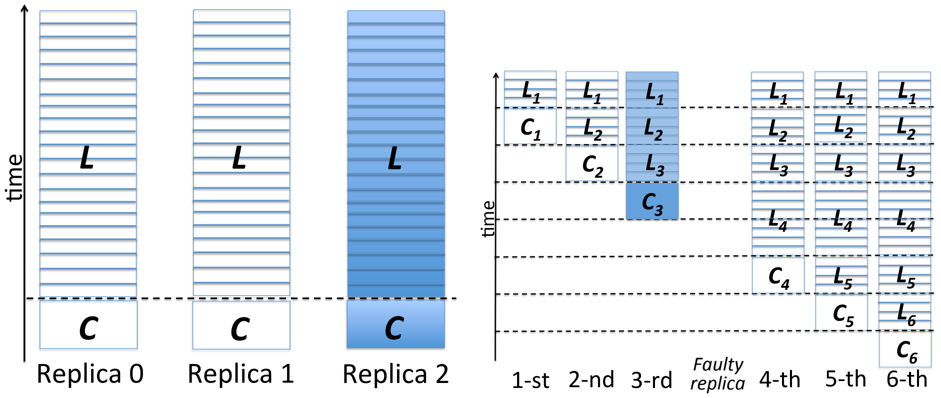
$$\begin{aligned} C_{max} &< \frac{1}{T_{max}} \times \left\lfloor \frac{P}{n} \right\rfloor \Longrightarrow \\ P &> n \times C_{max} \times T_{max} \end{aligned} \tag{2.1}$$

Equation 2.1 defines the minimum value for $P$ that can be used with sequential checkpoints. In our SCKV-Store example, for a state of 1GB and a 100% 4kB-write workload, we have $C_{max} \approx 15$s and $T_{max} \approx 4700$ ops/s, which means $P > 282000$. If more frequent checkpoints are required, the replicas can be organized in groups of at most $f$ replicas to take checkpoints together.

### 2.3.3 Collaborative State Transfer

The state transfer protocol is used to update the state of a replica during recovery, by transmitting log records ($L$) and checkpoints ($C$) from other replicas (see Figure 2.5(a)). Typically only one of the replicas returns the full state and log, while the others may just send a hash of this data for validation (only required in the BFT case). As showed in Section 2.2.3, this approach can degrade performance during recoveries. Furthermore, it does not work with sequential checkpoints, as the received state can not be directly validated with hashes of other replicas' checkpoints (as they are different). These limitations are addressed with the *collaborative state transfer* (CST) protocol.

Although the two previous techniques work both with crash-tolerant and BFT SMR, the CST protocol is substantially more complex with Byzantine faults. Consequently, we



(a) PBFT and others ($n = 4$).

(b) CST ($n = 7$).

Figure 2.5: Data transfer in different state transfer strategies.

start by describing a BFT version of the protocol (which also works for crash faults) and later, at the end of the section, we explain how CST can be simplified on a crash-tolerant system[2].

We designate by *leecher* the recovering replica and by *seeders* the replicas that send (parts of) their state. CST is triggered when a replica (leecher) starts (see Figure 2.6). Its first action is to use the local log and checkpoint to determine the last logged request and its sequence number (assigned by the ordering protocol), from now on called *agreement* $id$. The leecher then asks for the most recent logged agreement $id$ of the other replicas, and waits for replies until $n - f$ of them are collected (including its own $id$). The $ids$ are placed in a vector in descending order, and the largest $id$ available in $f + 1$ replicas is selected, to ensure that such agreement $id$ was logged by at least one correct replica (steps 1-3).

In BFT-SMaRt there is no parallel execution of agreements, so if one correct replica has ordered the $id$-th batch, it means with certainty that agreement $id$ was already processed by at least $f + 1$ correct replicas[3]. The other correct replicas, which might be a bit late, will also eventually process this agreement, when they receive the necessary messages.

Next, the leecher proceeds to obtain the state up to $id$ from a seeder and the associated validation data from $f$ other replicas. The active replicas are ordered by the freshness of the checkpoints, from the most recent to the oldest (step 4). A leecher can make this calculation based on $id$, as replicas take checkpoints at deterministic points, as explained in Section 2.3.2. We call the replica with $i$-th oldest checkpoint the $i$-th replica and the checkpoint $C_i$. The log of a replica is divided in segments, and each segment $L_i$ is the portion of the log required to update the state from $C_i$ to the more recent state $C_{i-1}$.

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

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

Therefore, we use the following notion of equivalence: $C_{i-1} \equiv C_i + L_i$. Notice that $L_1$ corresponds to the log records of the requests that were executed after the most recent checkpoint $C_1$ (see Figure 2.5(b) for $n = 7$).

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

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

If the check succeeds for $f$ replicas, the reached state is valid and the CST protocol can finish (step 13). If the validation fails, the leecher fetches the data from the $(f+2)$-th replica, which includes the log segments $L_1$, ..., $L_{f+2}$ and checkpoint $C_{f+2}$ (step 13 goes back to step 8). Then, it re-executes the validation protocol, considering as extra validation information the hashes that were produced with the data from the $(f+1)$-th replica (step 9). Notice that the validation still requires $f+1$ matching log segments and checkpoints, but now there are $f+2$ replicas involved, and the validation is successful even with one Byzantine replica. In the worst case, $f$ faulty replicas participate in the protocol, which requires $2f+1$ replicas to send some data, ensuring a correct majority and at least one valid state (log and checkpoint).

In the scenario of Figure 2.5(b), the 3rd replica (the $(f+1)$-th replica) sends $L_1$, $L_2$, $L_3$ and $C_3$, while the 2nd replica only transmits $HL_1 = \mathsf{H}(L_1)$, $HL_2 = \mathsf{H}(L_2)$ and $HC_2 = \mathsf{H}(C_2)$, and the 1st replica sends $HL_1 = \mathsf{H}(L_1)$ and $HC_1 = \mathsf{H}(C_1)$. The leecher next replays $L_3$ to get to state $C_3 + L_3$, and takes the intermediate checkpoint $C_2'$ and calculates the hash $HC_2' = \mathsf{H}(C_2')$. If $HC_2'$ matches $HC_2$ from the 2nd replica, and the hashes of log segments $L_2$ and $L_1$ from the 3rd replica are equal to $HL_2$ and $HL_1$ from the 2nd replica, then the first validation is successful. Next, a similar procedure is applied to replay $L_2$ and the validation data from the 1st replica. Now, the leecher only needs to replay $L_1$ to reach the state corresponding to the execution of request $id$.

While the state transfer protocol is running, replicas continue to create new checkpoints and logs since the recovery does not stop the processing of new requests. Therefore, they are required to keep old log segments and checkpoints to improve their chances to support the recovery of a slow leecher. However, to bound the required storage space,

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

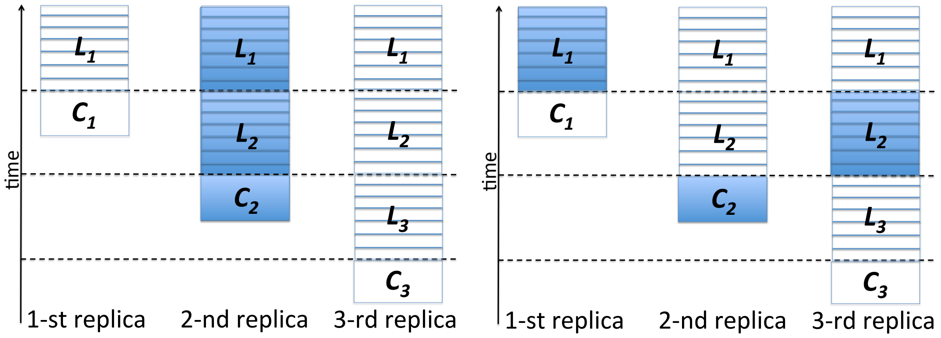
Figure 2.6: The CST recovery protocol called by the leecher after a restart. *Fetch* commands wait for replies within a timeout and go back to step 2 if they do not complete.

these old files are eventually removed, and the leecher might not be able to collect enough data to complete recovery. When this happens, it restarts the algorithm using a more recent request $id$ (a similar solution exists in all other state state transfer protocols that we are aware of, e.g., [13, 16]).

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

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

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



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

Figure 2.7: General and optimized CST with $f = 1$.

this state is recent, the largest agreement $id$ that is returned by $f + 1$ replicas is used.

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

**Optimizing CST for $f = 1$.** When $f = 1$ (and thus $n = 4$), a single recovering replica can degrade the performance of the system because one of $n - f$ replicas will be transferring the checkpoint and logs, delaying the execution of the agreements (as illustrated in Figure 2.7(a)). To avoid this problem, we spread the data transfer between the active replicas through the following optimization in an *initial recovery round*: the 2nd replica ($f + 1 = 2$) sends $C_2$ plus $\langle HL_1, HL_2 \rangle$ (instead of the checkpoint plus full log), while the 1st replica sends $L_1$ and $HC_1$ (instead of only hashes) and the 3rd replica sends $L_2$ (instead of not participating). If the validation of the received state fails, then the normal CST protocol is executed. This optimization is represented in Figure 2.7(b).

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

## 2.4 Final Remarks

This chapter discussed the performance problems that state management and state transfer can cause in a state machine replication protocol. Also, it proposes algorithms that solves the problems identified.

To provide durability in the SMR protocol, synchronous writes to stable storage would need to be performed. That can decrease the performance of the system as synchronous writes can take a long time to be performed and reduce the throughput of the system. We proposed an algorithm using parallel logging to alleviate the cost of log writing. The algorithm is composed of two parts. The first one consists of executing operations in parallel with the log writing. The second group requests in batches before processing them. Disks usually have a large bandwidth and a high access latency, so grouping operations can take advantage of that.

Another problem identified was that during the generation of a checkpoint by the replicas, the system stops to make progress as it can not process operations while it is reading the state. If the state of the application is big enough, to write the checkpoint to disk can take several seconds, leaving the application not responsive during the whole process. We defined an algorithm to have replicas taking checkpoints in different instants in time. As only one replica will be taking the checkpoint at a time, the system will have the quorum it needs to keep processing the operations.

The final problem we identified is the perturbations caused by the state transfer. When a replica asks the state to another replica, the replica supposed to send the state can take a lot of time reading the state from the disk and sending it through the network. We defined an algorithm where different replicas send parts of the state. The replica asking for the state combines the information received and updates its state. This can reduce the perturbation on each replica having a smaller decrease in the throughput of the system.

Next we will describe how we implemented the algorithms proposed in BFT-SMaRt and the evaluations performed to assess their efficiency.

# Chapter 3

# Dura-SMaRt

In order to validate our techniques, we extended the open-source BFT-SMaRt replication library [8] with a durability layer, placed between the request ordering and the service. We named the resulting system *Dura-SMaRt*, and used it to implement two typical SMR-based applications: a consistent key-value store and a coordination service.

In this chapter we describe the previous version of BFT-SMaRt and extensions made to implement the new techniques. We also describe the two applications and the evaluation performed to compare the results from the use of the new techniques with other approaches commonly used for state management and transfer.

## 3.1 BFT-SMaRt

BFT-SMaRt is a BFT SMR library developed using the Java language, offering a small and clear API to build clients and services on top of it. BFT-SMaRt was designed from the beginning with some principles intended to provide a robust yet efficient library. It provides a simple and extensible API with methods for clients to invoke operations on a service. On the server side the library provides several classes that can be extended to perform different operations like execution of ordered and unordered requests, batch of ordered requests and state management.

Protocols and building blocks of BFT-SMaRt were planned from the beginning to be implemented as independent modules, allowing development to focus on a specific module at a time. BFT-SMaRt has different threads executing services like message transport, ordering and execution. Executing these tasks in parallel reduces the latency of request processing with multiple clients, as the application can execute requests while messages from other clients are being ordered. BFT-SMaRt modules and their relationship are in Figure 3.1.

Clients using the library has to implement API methods to invoke ordered, unordered

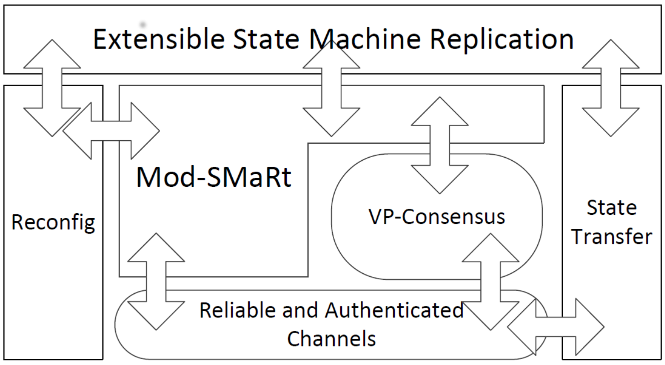

Extensible State Machine Replication
Mod-SMaRt
Reconfig
VP-Consensus
State Transfer
Reliable and Authenticated Channels

Figure 3.1: The modularity of BFT-SMaRt.

or asynchronous requests[1] to the application. In the server side the library provides different methods to be implemented by the application to execute the ordered and unordered requests from the clients. These methods allows the execution of a single ordered request, a batch of ordered requests and unordered requests. It also provides callback methods to take and install a snapshot of the application state.

### 3.1.1 State Management

After a batch or requests are ordered, BFT-SMaRt logs it in a sequential log before deliver it to the application. After the it is executed, the reply is returned to the client. When a replica finds itself to be delayed in relation to other replicas or a new replica is added to the system, it has to require a state transfer to update itself to the current state.

When a replica invokes a state transfer, it indicates a replica to send checkpoint plus the log of operations. The remaining replicas return a digest of the checkpoint plus the log of operations. After receiving a reply from the other replicas for the state transfer request, the recovering replica will validate the checkpoint with the digests to guarantee it is correct. The replica also will compare the different logs received. After install the checkpoint and replay the log operations the replica will be ready to process requests and participate again in the SMR protocol execution.

## 3.2 Implementation

### 3.2.1 Adding Durability to BFT-SMaRt

BFT-SMaRt originally offered an API for invoking and executing state machine operations, and some callback operations to fetch and set the service state. The implemented

[1] Ordered requests are totally ordered before delivered to the application. The client gets a reply only after the replicas executed and logged the request. Unordered requests are delivered directly to the replicas, skipping the ordering protocol. Unordered requests are not logged. Asynchronous requests may be requested as ordered or not. The difference from the previous requests is that the reply is sent to the client when received, without waiting for a quorum of replies.

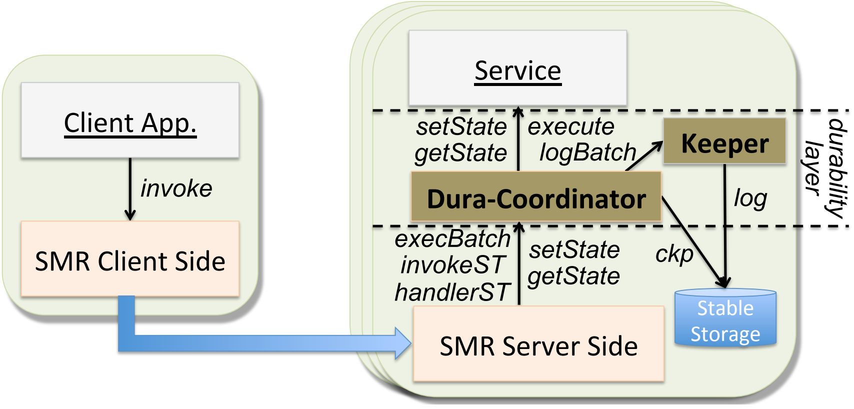


Figure 3.2: The Dura-SMaRt architecture.

protocols are described in [55] and follow the basic ideas introduced in PBFT and Aardvark [13, 18]. BFT-SMaRt is capable of ordering more than 80k 0-byte msg/s (the 0/0 microbenchmark used to evaluate BFT protocols [29, 40]) in our environment. However, this throughput drops to 20k and 5k msgs/s for 1kB and 4kB message sizes, respectively (the workloads we use – see Section 3.3).

We modified BFT-SMART to expose a *StateManager* interface to implement different state transfer strategies. We then moved the existing PBFT-like strategy to a *StandardStateManager* class and provided a new *DurableStateManager* class implementing the new techniques. The application developer can use one of the state transfer strategies defined or implement its own. The *Durability layer* presented in Figure 3.2 implements the new techniques described in Section 2.3 and uses the *DurableStateManager* class to handle the state transfer. Together with the changes in state transfer, we made the following modifications on BFT-SMaRt. First, we added a new server side operation to deliver batches of requests instead of one by one. This operation supplies ordered but not delivered requests spanning one or more agreements, so they can be logged in a single write by the *Keeper* thread. Second, we implemented the parallel checkpoints and collaborative state transfer in the *Dura-Coordinator* component, removing the old checkpoint and state transfer logic from BFT-SMaRt and defining an extensible API for implementing different state transfer strategies, as described above. Finally, we created a dedicated thread and socket to be used for state transfer in order to decrease its interference on request processing.

### 3.2.2 SCKV-store

The first application implemented with Dura-SMaRt was a *simple and consistent key-value store* (SCKV-Store) that supports the storage and retrieval of key-value pairs, alike to other services described in the literature, e.g., [19, 48]. The implementation of the SCKV-Store was greatly simplified, since consistency and availability come directly from SMR and durability is achieved with our new layer.

In the client side we exposed a subset of the methods available in the Java utilities interface `java.util.Map`. Clients can invoke operations in the replicated SCKV-Store object as if they were using a local Map instance. In the server side the data is stored in a `java.util.TreeMap`. We opted for this class instead of the `HashMap` to ensure that serialized versions of the same table (containing the same key-value pairs) are equal. The process of taking a checkpoint of the application state requires only the serialization and deserialization of the `TreeMap` object.

### 3.2.3 Durable DepSpace (DDS)

The second use case is a durable extension of the DepSpace coordination service [9], which originally stored all data only in memory. The system, named Durable DepSpace (DDS) [23], provides a tuple space interface in which tuples (variable-size sequences of typed fields) can be inserted, retrieved and removed. There are two important characteristics of DDS that differentiate it from similar services such as Chubby [11] and ZooKeeper [32]: it does not follow a hierarchical data model, since tuple spaces are, by definition, unstructured; and it tolerates Byzantine faults, instead of only crash faults.

DDS application data is stored in a structure containing configurable depths of the Java utilities `HasMap` class and a `Tuple` class containing the tuples. This is used to reduce the time necessary to find a tuple in the space. As an example, if a user creates a tuple containing `<A, B, C, D, E>` and has configured depth 2 in the application, the first two fields are stored in maps. When the user searches for tuple by any of the two first fields, the tuples are searched in the map instead of iterating over all the tuples. In the scenario described, the tuple would be stored in a structure as `HashMap(A, (HashMap (B, Tuple<A, B, C, D, E>))`.

Snapshots in DDS were implemented as a log containing one insert operation for each tuple in the space, in this way the same code for executing the logged operations can be used for processing snapshots. Also, as the data structure uses *serializable* Java classes, it was not necessary to write additional code to serialize and deserialize the objects.

The addition of durability to DepSpace basically required the replacement of its original replication layer by Dura-SMaRt.

## 3.3 Evaluation

This section evaluates the effectiveness of our techniques for implementing durable SMR services. In particular, we devised experiments to answer the following questions: (1) What is the cost of adding durability to SMR services? (2) How much does *parallel logging* improve the efficiency of durable SMR with synchronous disk and SSD writes? (3) Can *sequential checkpoints* remove the costs of taking checkpoints in durable SMR?

(4) How does *collaborative state transfer* affect replica recoveries for different values of $f$? Question 1 was addressed in Section 1.2, so we focus on questions 2-4.

### 3.3.1 Case Studies and Workloads

As already mentioned, we consider two SMR-based services implemented using Dura-SMaRt: the SCKV-Store and the DDS coordination service. Although in practice, these systems tend to serve mixed or read-intensive workloads [19, 32], we focus on write operations because they stress both the ordering protocol and the durable storage (disk or SSD). Reads, on the other hand, can be served from memory, without running the ordering protocol. Therefore, we consider a 100%-write workload, which has to be processed by an agreement, execution and logging. For the SCKV-Store, we use YCSB [19] with a new workload composed of 100% of replaces of 4kB-values, making our results comparable to other recent SMR-based storage systems [10, 52, 60]. For DDS, we consider the insertion of random tuples with four fields containing strings, with a total size of 1kB, creating a workload with a pattern equivalent to the ZooKeeper evaluation [32, 35].

### 3.3.2 Experimental Environment

All experiments, including the ones in Section 2.1, were executed in a cluster of 14 machines interconnected by a gigabit ethernet. Each machine has two quad-core 2.27 GHz Intel Xeon E5520, 32 GB of RAM memory, a 146 GB 15000 RPM SCSI disk and a 120 GB SATA Flash SSD. We ran the IOzone benchmark[2] on our disk and SSD to understand their performance under the kind of workload we are interested: rewrite (append) for records of 1MB and 4MB (the maximum size of the request batch to be logged in DDS and SCKV-Store, respectively). The results are presented in Table 3.1.

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

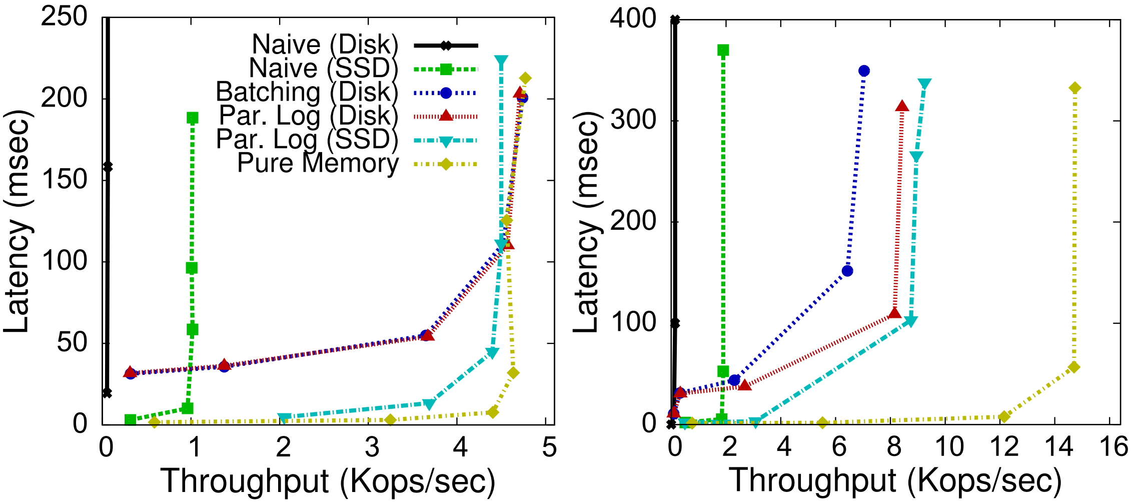
Table 3.1: IOZone microbenchmark on the employed disk and SSD.

### 3.3.3 Parallel Logging

Figure 3.3(a) displays latency-throughput curves for the SCKV-Store considering several durability variants. The figure shows that naive (synchronous) disk and SSD logging achieve a throughput of 63 and 1017 ops/s, respectively, while a pure memory version with no durability reaches a throughput of around 4772 ops/s.

Parallel logging involves two ideas, the storage of batches of operations in a single write and the execution of operations in parallel with the secondary storage accesses. The

[2] `http://www.iozone.org`.



(a) SCKV-Store.

(b) Durable DepSpace.

Figure 3.3: Latency-throughput curves for several variants of the SCKV-Store and DDS considering 100%-write workloads of 4kB and 1kB, respectively. Disk and SSD logging are always done synchronously. The legend in (a) is valid also for (b).

use of batch delivery alone allowed for a throughput of 4739 ops/s with disks (a 75× improvement over naive disk logging). This roughly represents what would be achieved in Paxos [39, 42], ZooKeeper [32] or UpRight [17], with requests being logged during the agreement protocol. Interestingly, the addition of a separated thread to write the batch of operations, does not improve the throughput of this system. This occurs because a local put on SCKV-Store replica is very efficient, with almost no effect on the throughput.

The use of parallel logging with SSDs improves the latency of the system by 30-50ms when compared with disks until a load of 4400 ops/s. After this point, parallel logging with SSDs achieves a peak throughput of 4500 ops/s, 5% less than parallel logging with disk (4710 ops/s), with the same observed latency. This is consistent with the IOzone results, in which the data throughput of our disk is better than SSDs for big records (see Table 3.1). Overall, parallel logging with disk achieves 98% of the throughput of the pure memory solution, being the replication layer the main bottleneck of the system. Moreover, the use of SSDs neither solves the problem that parallel logging addresses, nor improves the performance of our technique, being thus not effective in eliminating the log bottleneck of durable SMR.

Figure 3.3(b) presents the results of a similar experiment, but now considering DDS with the same durability variants as in SCKV-Store. The figure shows that a version of DDS with naive logging in disk (resp. SSD) achieves a throughput of 143 ops/s (resp. 1900 ops/s), while a pure memory system (DepSpace), reaches 14739 ops/s. The use of batch delivery improves the performance of disk logging to 7153 ops/s (a 50× improvement). However, differently from what happens with SCKV-Store, the use of parallel logging in disk further improves the system throughput to 8430 ops/s, an improvement of

18% when compared with batching alone. This difference is due to the fact that inserting a tuple requires traversing many layers [9] and the update of an hierarchical index, which takes a non-negligible time (0.04 ms), and impacts the performance of the system if done sequentially with logging. The difference would be even bigger if the SMR service requires more processing. Finally, the use of SSDs with parallel logging in DDS was more effective than with the SCKV-Store, increasing the peak throughput of the system to 9250 ops/s (an improvement of 10% when compared with disks). Again, this is consistent with our IOzone results: we use 1kB requests here, so the batches are smaller than in SCKV-Store, and SSDs are more efficient with smaller writes (see Table 3.1).

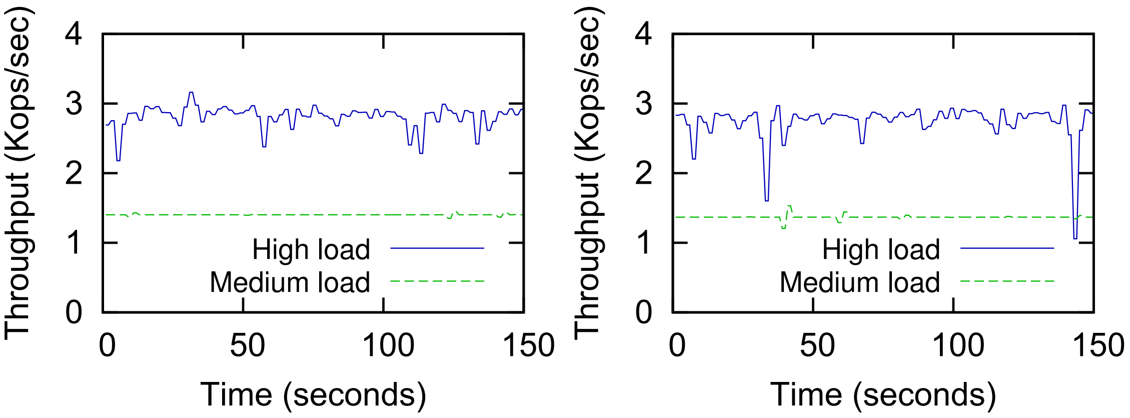
Notice that DDS could not achieve a throughput similar to a pure memory system. This happens because, as discussed in Section 2.3.1, the throughput of parallel logging will be closer to a pure memory system if the time required to process a batch of requests is akin to the time to log this batch. In the experiments, we observed that the workload makes BFT-SMaRt deliver batches of approximately 750 requests on average. The local execution of such batch takes around 30 ms, and the logging of this batch on disk entails 70 ms. This implies a maximum throughput of 10.750 ops/s, which is close to the obtained values. With this workload, the execution time matches the log time (around 500 ms) for batches of 30K operations. These batches require the replication library to reach a throughput of 60K 1kB msgs/s, three times more than what BFT-SMaRt achieves for this message size.

### 3.3.4 Sequential Checkpointing

Figure 3.4 illustrates the effect of executing sequential checkpoints in disks with SCKV-Store[3] during a 3-minute execution period.

When compared with the results of Figure 2.2 for synchronized checkpoints, one can observe that the unavailability periods no longer occur, as the 4 replicas take checkpoints separately. This is valid both when there is a high and medium load on the service and with disks and SSDs (not show). However, if the system is under stress (high load), it is possible to notice a periodic small decrease on the throughput happening with both 500MB and 1GB states (Figures 3.4(a) and 3.4(b)). This behavior is justified because at every $\lfloor \frac{P}{n} \rfloor$ requests one of the replicas takes a checkpoint. When this occurs, the replica stops executing the agreements, which causes it to become a bit late (once it resumes processing) when compared with the other replicas. While the replica is still catching up, another replica initiates the checkpoint, and therefore, a few agreements get delayed as the quorum is not immediately available. Notice that this effect does not exist if the

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



(a) 500MB state.

(b) 1GB state.

Figure 3.4: SCKV-Store throughput with sequential checkpoints with different write-only loads and state size.

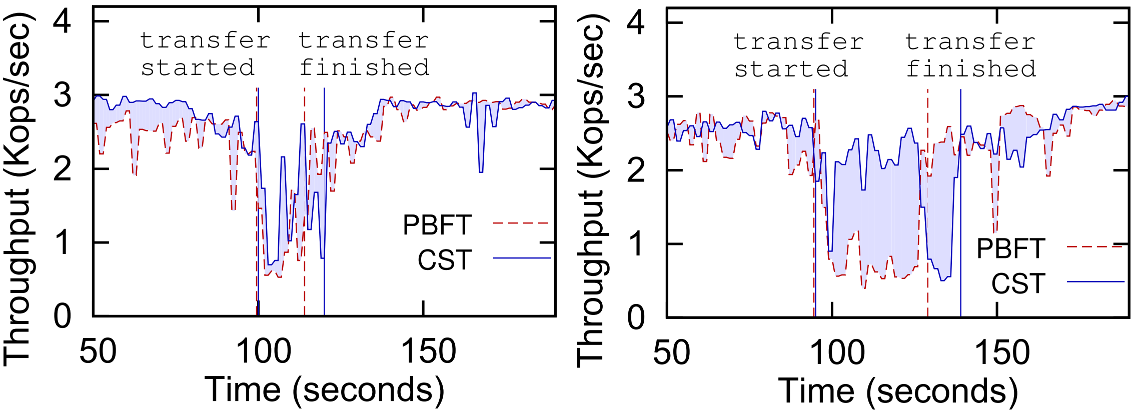
system has less load or if there is sufficient time between sequential checkpoints to allow replicas to catch up ("Medium load" line in Figure 3.4).

### 3.3.5 Collaborative State Transfer

This section evaluates the benefits of CST when compared to a PBFT-like state transfer in the SCKV-Store with disks, with 4 and 7 replicas, considering two state sizes. In all experiments a single replica recovery is triggered when the log size is approximately twice the state size, to simulate the condition of Figure 2.7(b).

Figure 3.5 displays the observed throughput of some executions of a system with $n = 4$, running PBFT and the CST algorithm optimized for $f = 1$, for states of 500MB and 1GB, respectively. A PBFT-like state transfer takes 30 (resp. 16) seconds to deliver the whole 1 GB (resp. 500MB) of state with a sole replica transmitter. In this period, the system processes 741 (resp. 984) ops/sec on average. CST optimized for $f = 1$ divides the state transfer by three replicas, where one sends the state and the other two up to half the log each. Overall, this operation takes 42 (resp. 20) seconds for a state of 1GB (resp. 500MB), $28\%$ (resp. $20\%$) more than with the PBFT-like solution for the same state size. However, during this period the system processes 1809 (resp. 1426) ops/sec on average. Overall, the SCKV-Store with a state of 1GB achieves only $24\%$ (or $32\%$ for 500MB-state) of its normal throughput with a PBFT-like state transfer, while the use of CST raises this number to $60\%$ (or $47\%$ for 500MB-state).

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



(a) 500MB and $n = 4$.

(b) 1GB and $n = 4$.

Figure 3.5: Effect of a replica recovery on SCKV-Store throughput using CST with $f = 1$ and different state sizes.

throughput (60% of the normal throughput), even considering the extra time required for fetching the state (+28%).

We did an analogous experiment for $n = 7$ (see Figure 3.6) and observed that, as expected, the state transfer no longer causes a degradation on the system throughput (both for CST and PBFT) since state is fetched from a single replica, which is available since $n = 7$ and there is only one faulty replica (see Figure 2.5). We repeated the experiment for $n = 7$ with the state of 1GB being fetched from the leader, and we noticed a 65% degradation on the throughput.

A comparable effect occurs if the state is obtained from the leader in CST. As a cautionary note, we would like to remark that when using spare replicas for "cheap" faulty recovery, it is important to avoid fetching the state from the leader replica (as in [11, 16, 32, 52]) because this replica dictates the overall system performance.

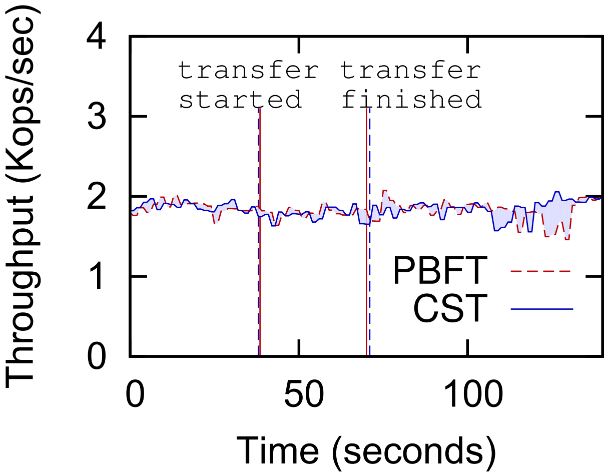


Figure 3.6: Effect of a replica recovery on SCKV-Store throughput using CST with $f = 2$ and 1GB state size.

## 3.4 Final Remarks

This section described the effort to implement the techniques proposed in Section 2 in the BFT-SMaRt replication library. The current version of BFT-SMaRt available at `http://code.google.com/p/bft-smart` contains the durability layer that can be activated via configuration files.

This chapter also included a detailed evaluation of the impact that the new techniques have in the protocol performance when compared to the previous version. All three new techniques proposed were proven effective during the evaluation performed with the two use cases - SCKV - Store and Durable DepSpace.

We used Dura-SMaRt as the replication layer of a database replication middleware called SteelDB, which we present next. SteelDB uses Dura-SMaRt to successfully manage communication between replicas, and also store and transfer state to recovering replicas. We did not need to perform any changes in Dura-SMaRt code to serve as SteelDB replication layer, proving the protocol to be robust enough to serve complex applications.

# Chapter 4

# A Byzantine Fault-Tolerant Transactional Database

## 4.1 Introduction

Database managements systems store and provide access to data to multiple concurrent users. Usually, this data is stored in tables and is updated or queried using a standard query definition language (SQL). Several different vendors and open-source projects implements DBMS products. Such products may differ in several concepts like concurrency control, transaction management, relationship enforcements and extra tools provided by vendors.

To manage access from concurrent users to the same database, enforcing integrity of data, transactions are used. Transactions are boundaries applied to a group of operations executed over a database. A transaction must enforce ACID properties (atomicity, consistency, isolation and durability) [28]. Atomicity means that all operations must be executed or none is. In a case of failure during the transaction, the database has to be in the same state it was before. Consistency guarantees that all operations inside a transaction will bring the database from one valid state to another. Constraints and relationships should be respected through the entire transaction. Isolation expects that no change performed from inside a transaction is visible outside it until it is finished. This is enforced by the concurrency control mechanism employed. Durability enforces that all changes performed stays visible after the end of a transaction, even in the event of crashes and power outages.

To isolate the operations executed inside a transaction, multi-version concurrency control (MVCC) can be used to allow several users to execute read and write operations on values without the need for serialization. Serialization [6] of transactions usually requires locking on tables and rows, sometimes blocking read operations without need, reducing the concurrency and performance of the system. Snapshot Isolation [5], one of the isolation levels inside MVCC, manages concurrency by taking snapshots of the state and numbering versions between changes. A write operation, executed from inside a transaction, will update the value and increment the version when the transaction finishes. If the value was updated from another transaction, the update fails. When an operation reads a value, it first checks the version number of the object. When the transaction is about to be finished (by a *commit* operation), the object is checked again to validate if it was not modified after the read.

Several commercial and open-source DBMSs from different vendors like Oracle, MySQL and PostgreSQL employ replication to database servers. The replication schemas are designed for the specific vendor implementation to provide resilience and availability in the presence of faults. While this can provide availability and some degree of resilience for crash faults, the client is tied to that specific vendor DBMS distribution. Also, such schema is limited by the efficiency of mechanisms of failover/failback to detect failures and switch masters [15].

An alternative to that approach is the use of a database replication middleware [25, 57, 34]. A DBMS middleware is placed between the client and the database servers. Instead of make the request to the DBMS directly, clients request operations to the middleware that serializes requests and forward them to multiple servers providing fault tolerance.

According to [15] a DBMS replication middleware may face several issues sometimes neglected during its design. Database internals like temporary tables, and stored procedures and also database specific idioms may difficult diverse replication with different vendors implementations and even versions. The use of non-deterministic functions like random number generators or current timestamp may cause inconsistency between states.

## 4.2 Byzantium

Byzantium considers all operations executed against databases to run inside transactions. To increase concurrent execution of transactions, Byzantium assumes that DBMS implementation provides snapshot isolation.

Transactions in Byzantium are flagged as read-write or read-only. By the time a transaction is created it is considered read-only. Transactions will be promoted to read-write when the first write message within the transaction is issued.

Byzantium is a database replication middleware tolerant to Byzantine faults. It uses PBFT [13] as the replication library requiring $3f+1$ replicas to tolerate $f$ faults. Byzantium architecture is presented in Figure 4.1.

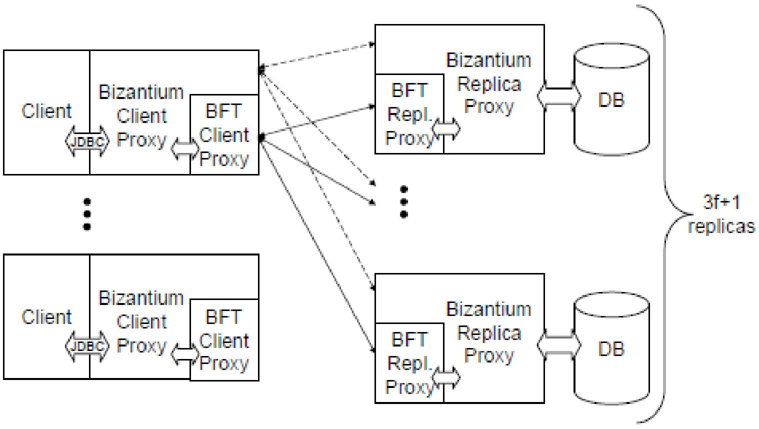


Figure 4.1: The Byzantium architecture [25].

To reduce the cost of replication, operations received in read-only transactions are sent only to $f+1$ replicas. The Byzantium client waits for $f$ replies before return the result to the client. By the time of commit it verifies if the $f$ replies matches the remaining reply before confirming the transaction. If results diverge, the operation is executed in the remaining replicas to validate the result. If $f+1$ matching replies cannot be confirmed, the transaction is aborted.

Operations that occur in a read-write transaction are sent to all replicas but only executed in one of them, the master replica. In this way the client do not need to wait for a reply quorum to confirm the operation. During the processing of a transaction, the client will keep a list of operations sent to the master and the results returned. By the time of commit, the client sends the list of operations and results to all replicas that if the operations were correctly executed. If the operations match, the replicas will execute all operations and compare the results with results informed from the master. If the results match the commit will be confirmed. Otherwise, the transaction will be rolled back and the client will be informed, so that it can proceed with a suspect master operation.

Byzantium defines also a suspect master protocol to change a faulty master. When a client notices that the master returned incorrect results or took longer than a predefined timeout to return results, it sends a "suspect master" message to the replicas. The replicas try then to confirm with the master if it received the messages from the client. If confirmed that the master is faulty, the replicas define the replica with the next id as the master and will inform the client about it. Open transactions will be aborted and the clients will be informed about that.

Byzantium defines two versions of the protocol, called single-master and multi-master. In the single-master version, all transactions consider the same replica to be the master. In the multi-master protocol, at the beginning of a transaction one of the replicas is randomly selected to be the master replica. In that case, each transaction will have a master and this master can be different from other transactions. According to evaluation performed in the Byzantium paper [25], the single-master performs better in read-write dominated workloads while the multi-master version performs better in read-only dominated workloads. This may be explained by the fact that in the multi-master version a client can choose a master in the same network it is, reducing the communication time to the replica. As messages are executed only in the master before commit, it will have only to send the *begin* and *commit* messages to all the replicas.

## 4.3 SteelDB

SteelDB is a middleware to replicate data, providing resilience for database management systems, assuming that a predefined number of replicas can be faulty. The source code is available at `http://code.google.com/p/steeldb`. SteelDB borrows some ideas from Byzantium, like the optimistic execution of transactions in the master and the master change protocol. The SteelDB client is a driver implementing the JDBC [33] specification. Clients already using a JDBC driver to execute operations in a database need only to change its configuration to use our driver instead. Although our design uses some ideas from Byzantium, we have some differences from it. We implemented only the single-master version of the Byzantium, as we decided to focus our efforts in provide a functional and efficient state transfer and master change protocols. Another difference is that Byzantium uses PBFT [13] as the replication layer, while we use BFT-SMaRt to replicate data and manage state transfer.

Each replica of SteelDB is a client of a database server instance, in which it executes the operations delivered to it. The architecture of SteelDB is presented in Figure 4.2.

The clients of SteelDB implement the JDBC specification to invoke operations in BFT-SMaRt service proxy. We take advantage of the use of transactions to reduce the number of messages exchanged between clients and replicas. Unlike in Byzantium, when an operation is inside a transaction, the client sends it only to the master replica. The master executes the operation and returns the result to the client. When the client tries to commit the transaction, it sends the list of operations executed and responses received during the transaction to all replicas. The replicas execute the operations and compare the results with results returned from the master. This makes the operations simpler to execute but pushes all the validation work to the commit operation.

When a client issues a request to SteelDB client, it is replicated to multiple servers. The request is delivered to the replica by invoking the callback method *invokeOrdered*. Depending on the type of request, it may perform different operations like open a connection to the database, execute a query or update and commit a transaction. In the Figure 4.3 we present a sequence

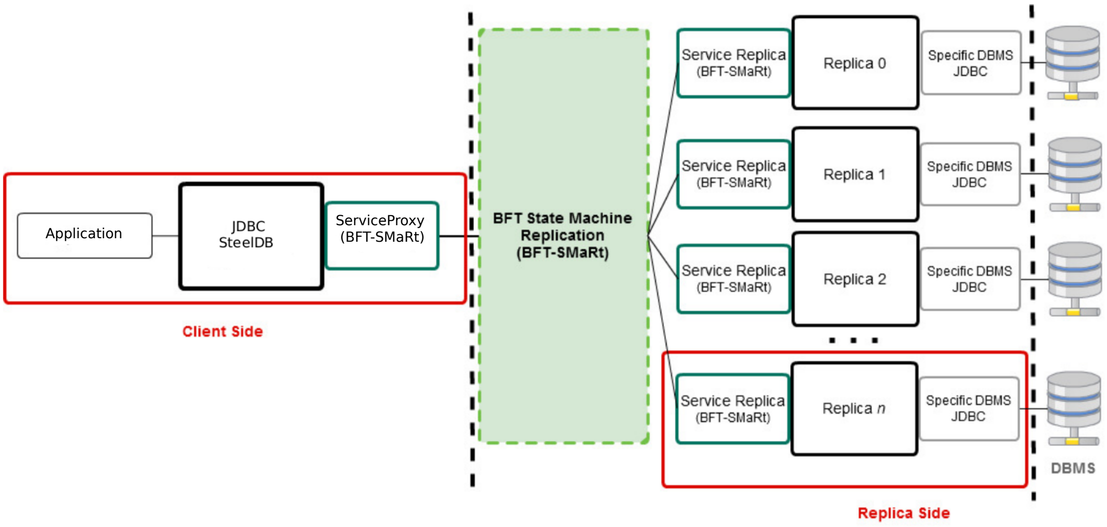


Figure 4.2: The SteelDB architecture.

diagram describing the process of execute an update operation in the database and commit the open transaction.

The processing of the request starts when the `invokeOrdered` callback method is invoked by BFT-SMART. That method checks the request and call `executeUpdate` in the `MessageProcessor` class. That class contains methods for all operations to be executed in the database. After `MessageProcessor` receives the request, it gets a `ConnectionManager` from a `SessionManager` class. `SessionManager` contains a map with all open connections from clients, along with the client identification number. The `ConnectionManager` contains information about the client session, like connection parameters, the list of requests processed in the open transaction and flags about the connection state. It also contains the JDBC Connection to the database, in which the operations are executed.

After the client gets the `ConnManager` object it opens a `Statement` object to the database, in which the database request is generated, using the SQL command and the parameters informed. After the statement is filled with the information, it is executed in the database, using the database connection.

That finishes the request execution but if the database connection is not configured as auto-commit, it is not persisted yet. When the user performs a commit request, it follows the same path as the `executeUpdate` method call, with the difference that instead of have a statement created and executed, it calls a commit operation using the database connection.

In the following subsections we describe some relevant issues we had to address to implement SteelDB on top of BFT-SMART with the durability layer described in previous chapters.

### 4.3.1 Enforcing FIFO Order

All previous services developed over BFT-SMART supported simple read, write or read-write operations, without any support for ACID transactions. In this scenario, there was no problem if some unordered read operation arrived out of order in $f$ replicas. This happens because the client expects a number of replicas to execute operations and return matching results before sending its next request.

The problem is, with a transactional protocol, if an operation within a transaction arrives at some replica before the transaction BEGIN (or the database connection is established), it will be

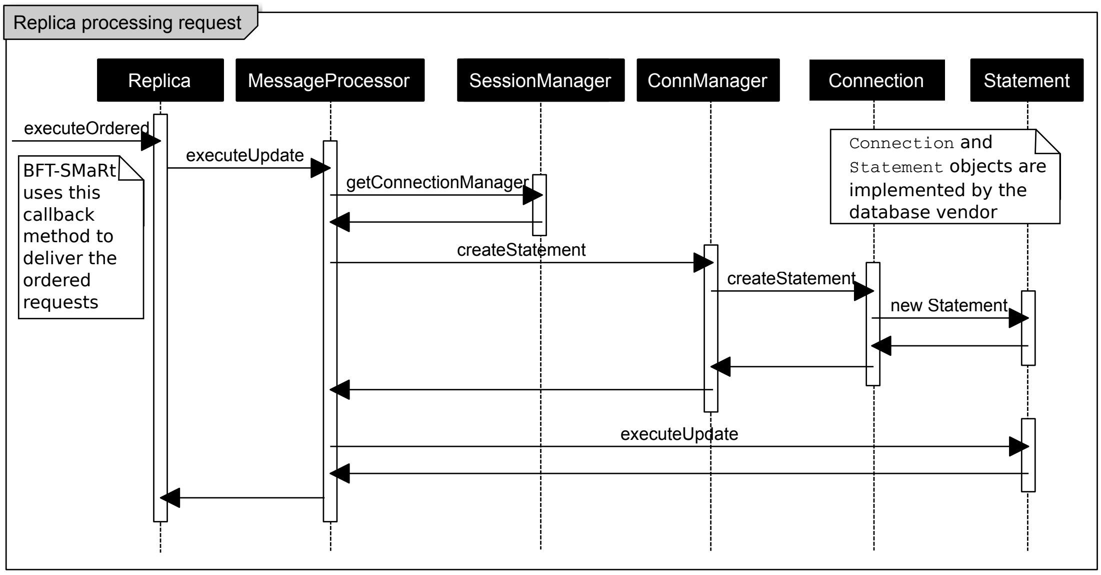


Figure 4.3: Sequence diagram of a request processinb by a SteelDB replica.

rejected, and the client may observe problems. The reason for this is that BFT-SMARt either considers total order requests or unordered requests, there is no mechanism ensuring FIFO order, as required in Byzantium [25]. To deal with this limitation without modifying BFT-SMARt we devised an ordering layer that uses application-level sequence numbers (defined in our JDBC driver and verified in the replica' service) and locks to ensure messages are processed in the order that they are sent by a client. After several tests we discovered that this would not work without changing the internal handling of requests in BFT-SMARt.

We found that it would be easier to change the protocol to optimistically execute operations inside transactions only in the master replica and send them to the other replicas only before commit (as explained in the previous section). This removed the requirement of execute clients messages in FIFO order.

### 4.3.2 Issues with the JDBC Specification

The transaction management in JDBC is slightly different from the way that a DBMS operates. Usually, the sequence of steps to process a transaction in a database includes: open a connection, begin a transaction, execute the operations and commit or rollback the transaction. The description of Byzantium algorithm follows this sequence of operations.

JDBC manages the transaction boundaries transparently. When a user opens a connection to the database, the connection is opened in auto-commit mode. This means that all operations will be reflected in the database immediately, without need for an explicit commit operation. If the client needs to create a transaction, it has to change the connection auto-commit flag to false. This has the same effect as issuing a begin command to the database. Operations are executed in the database and the transaction is finished by a commit or an abort (usually called rollback). After a commit, or a rollback, the auto-commit flag is not changed, which means that the connection is still not in auto-commit mode, so, one transaction was committed and another started. A client can also finish a transaction by setting again the auto-commit flag to true. It will commit the current transaction before the change.

We had to manage client operations to store the auto-commit flag for all operations. Byzantium also defines that transactions should not be considered read-write before the first update operation is issued. To implement that we needed to add another flag to each connection to define if it has a read-only or read-write transaction. By the time a transaction is promoted to read-write, all operations are executed only in the master replica. In the other replicas operations are executed only during a commit.

When a JDBC connection has the auto-commit flag set to true, all operations, even if the read-only (i.e., `SELECT`) are totally ordered though replicas. If the flag is false, the only messages that are totally ordered are the begin (set auto-commit false) and commit or rollback.

### 4.3.3 State Transfer

State machine replication assumes that a replicated service will make progress even in the presence of up to $f$ faulty replicas. This means that situations may occur when up to $f$ replicas are not in the same state of the others. Also there may be times when a replica crashes and recovers after some time. As the system continues to make progress, the recovered or late replica may not have an up to date copy of the state, so it has to ask other replicas to provide the state. This recovery process is managed by the state transfer protocol.

Without the use of a state transfer protocol, after $f$ replicas are compromised, the system is vulnerable to attacks or crashes. At that point if any additional faults occur, the integrity of the system cannot be confirmed, as there is no minimum quorum of participants to validate operations.

SteelDB uses the Dura-SMaRt (described in the last two chapters) for managing state transfer. Following the new state transfer strategy, when a replica starts, the first action is to load state from durable media and check with the other replicas if they processed the same operations (execution ids). If this is true, the replica is ready for execution and waits for requests from the clients. Otherwise, the replica considers itself late and issues a request to other replicas asking for the state.

One replica is defined to send the state and the others will send digests of the state. After receiving the state and digests, the replica can compare the data to find if the state is correct. If confirmed, the state is installed and the replica continues to process the remaining requests. The state transfer protocol recovers a faulty replica, enabling it to resume request processing.

Byzantium defines an optimization where replicas store operations to be executed before commit. We tried to implement this protocol, but it added a lot of complexity to the state transfer. We had to manage the state of the application plus partial data that was not stored in the database but could not be ignored. This data needed to be executed in the new replica after the state was restored.

To solve this problem, in SteelDB we execute requests that are inside a read-write transaction optimistically in the master. During commit (or rollback) we totally order the requests through the replicas and log operations to disk. Using this strategy, we can use the state transfer protocol defined in BFT-SMaRt (more precisely, the version implementing Dura-SMaRt) without changes. BFT-SMaRt state transfer protocol defines a strategy to take copies of the application state to be transferred to replicas that may request it during the protocol execution (see Section 3.2). This strategy comprises the log of all operations that changes the state of the application. To prevent the log from growing indefinitely , in pre defined periods, the application state is saved in a serialized format called checkpoint and the log is erased. In SteelDB we ask the DBMS to generate a dump of the database information during checkpoint periods. Together with the dump we store the information about open connections to be restored in the new replicas.

When a replica requests a state transfer, it receives the state from other replicas, comprised of the checkpoint and log of operations. The checkpoint contains the database dump plus information about client connections. After receiving the state, the recovering replica installs the dump, and opens the connections necessary to perform the requests. After replaying the requests, the replica is ready to resume processing.

When a recovering replica asks the state from other replicas it receives the dump of the database together with information about connections. It can then restore the database, open the connections and execute the logged operations to update itself to the state it received the first request after start.

As the database dump is managed by the DBMS, the idiom of the queries created during the dump may vary from vendor to vendor. Vendors may also include comments and change the order of operation to better suits their needs. That can add a lot of complexity to the process of comparing dumps from different vendors for authenticity.

The work on [57] provides diversity of database vendors by the use of a module to translate queries to a common idiom before and after sending it to database. During the state transfer this translator module will process the database dump to translate all the queries. Although we started working on a similar module to translate queries, we decided to focus our work on the efficient management of state transfer and guarantee that SteelDB can restore both state and user sessions for recovering replicas.

### 4.3.4 Master Change

Byzantium defines a master change protocol [25] that works as follows: when the master replica takes more than a predefined timeout to process a request, the client informs the other replicas about the presence of a faulty master in a ”master change” request. Together with this request, the client sends the operations executed so far in the current transaction. When the replicas receive the request for a master change operation, they will update their master id deterministically to a new one. The new master will execute the requests informed by the client and reply with the results.

To prevent a malicious client from request several master changes, the replicas logs the requests sent from clients. If a client tries to request multiple master changes in less then a predefined time, the requests are ignored.

After a master change, clients will not know about the new master until the execution of the next operation. If the next operation is sent only to the old master, it will fail, as the old master is offline. When the client requests the master change, replicas will know that the change was already executed and the client will be informed. If instead of send a message only to the master, the client send a message to all replicas (commit message or message in an auto-commit connection) the operation will be executed and the replicas will inform the new master together with the reply to the client.

When the old master recovers, it may still consider itself the current master. To prevent problems, every message exchange between replicas includes the master id. When the old master sends a message to other replicas it will get a quorum of replies with the current master id (that is not itself) and will store that information.

### 4.3.5 Optimizations

Byzantium defines optimizations in the protocol to increase performance in the presence of read-dominated workloads.

Messages executed on read-only transactions can be executed speculatively in only two replicas and the first result is returned to client. Only by the time of commit the second result is compared. This works when the operation is executed inside a transaction. If the connection is in auto-commit mode, the message has yet to be validated by a quorum of replicas, as there is no commit time to invalidate the response.

As described before, the storage of operations in replicas before commit would increase the complexity of the state transfer protocol. Instead, we decided to execute operations only in the master replica and execute the whole transaction before commit in the other replicas. If the replicas are distributed across different networks the latency in communication between the client and the replica can be reduced by having the initial master as the replica in the same network of the majority of the clients.

## 4.4 Final Remarks

We wanted to develop SteelDB as a complex use case on top of BFT-SMaRt. We borrowed most of the ideas in SteelDB from the work present on the Byzantium paper [25]. The main difference from our work to Byzantium is that ours focuses in having a resilient framework through the use of the efficient state transfer provided by Dura-SMaRt, while Byzantium focuses on performance of operations execution by the use of several techniques like multiple masters and optimized execution of messages though different quorum of replicas.

We implemented SteelDB using Dura-SMaRt as a blackbox replication layer, attesting that the library offers a service that can be implemented without changes. We used SteelDB as a database replication middleware for a use case in the TClouds project [3]. We performed several functional and performance tests that we describe in the next chapter.

# Chapter 5

# SteelDB Evaluation

This chapter describes the evaluation of SteelDB. This evaluation is centered on one of the use cases of the Tclouds FP7 project.

## 5.1 Evaluation Context and Environment

SteelDB was used as a component in an integration between partners for a use case scenario of the TClouds project [3]. The TClouds was a project co-founded by the European Commission, including several partners like FCUL, Sirrix, EDP, Efacec and IBM Research Zurich. The mission of TClouds was to develop an advanced cloud infrastructure delivering computing and storage that achieves a new level of security, privacy, and resilience being cost-efficient, simple, and scalable.

TClouds had as one of its use cases a Smart Lighting System (SLS) scenario. SLS [49] is a critical web application that maintains a schedule of events (e.g., turn lights on/on) for each equipment of one or more urban districts. The system end-users access it through a web interface to define new events or view the schedule of events for each area of a municipality. The SCADA system that controls the managed equipment accesses such web service to obtain possible changes on the schedule of events for devices.

The SLS application was hosted by a Tomcat [1] application server. The application uses the Hibernate ORM (Object-Relational Mapping) [2] to perform the mapping between model objects and the database queries. It uses the database implementation of the JDBC specification to connect to the database.

In the final year of TClouds an evaluation of the integration between components developed by different partners was performed. We replicated the database layer of the Smart Lighting System to provide dependability to the service. This replication was done on top of a hybrid architecture containing a private cloud provided by one of the partners and VMs in two public clouds. The architecture is represented in Figure 5.1. The private cloud, called TrustedInfrastructure Cloud (TC) contained two application servers for the SLS application simulating two main data centers. The private cloud contained also two servers with a SteelDB replica and an H2 database each. The two remaining SteelDB replicas, along with the H2 database server were deployed on public clouds, one in Amazon EC2 and the other in Microsoft Azure.

In that architecture we simulate two main data centers, one in Lisbon and another in Porto. Both servers serve requests to the SLS application at the same time. To reduce the delay in communication, we placed the SteelDB master replica in one of the data centers in TC. The master is switched between both servers in the case of a master change.

The TrustedInfrastructure Cloud was served from an Intel Xeon CPUE5630 @ 2.53GHz machine, with 8 cores and 16GB of RAM memory. In that machine we had six virtual machines

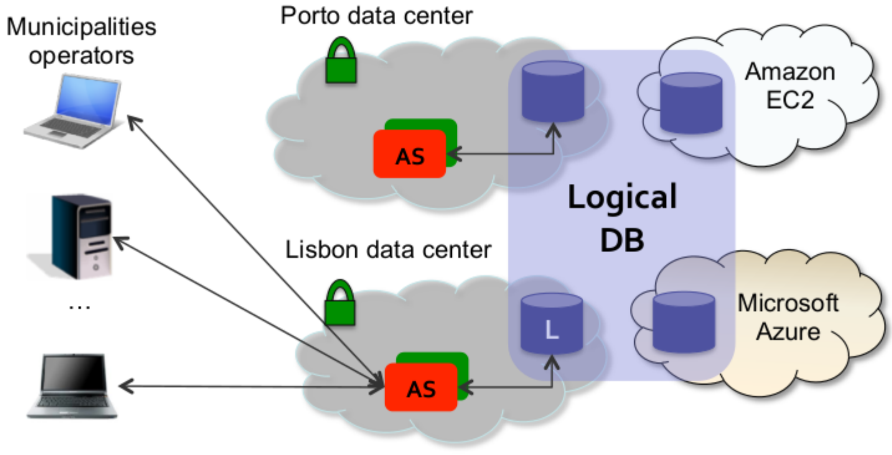


Figure 5.1: The Smart Lighting System Architecture.

(VMs) with 2GB of RAM memory each. Two VMs were used for the Tomcat application servers and four for the SteelDB replicas, containing one SteelDB replica and a H2 database instance each. Notice that in baseline tests without SteelDB or tests where we had only two SteelDB replicas in the TC, we didn't use all the six VMs at the same time. The virtual machine in Amazon EC2 was a 64bits M1 Standard Medium configuration with one vCPU and 3.75GB of RAM memory. The virtual machine in Microsoft Azure was an M2 Medium configuration with two virtual cores and 3.5GB of RAM memory.

Servers in the TrustedInfrastructure Cloud are considered secure by mechanisms like access only from attested clients and over a trusted virtual domain. As we have the SteelDB master always in one of the two TC data centers, that allowed us to add an optimization in SteelDB to perform read requests only in the master replica. This version optimized for read queries was only used during the evaluation activities, in the scenarios described next. We did not include this optimization in the code available in SteelDB page (`http://code.google.com/p/steeldb`), as we can not assume that replicas are secure in common environments. As we show next, the read optimization provided results similar to the write requests being executed in the master before commit.

## 5.2 Integration Scenarios

The TClouds project contains several use case scenarios, set for validating the Smart Lighting System [49] integration with other components. Here we describe only the two that are relevant for SteelDB.

The two main evaluation activities tested the functionality and performance of SteelDB with and without the presence of failures. Both activities performed multiple executions of different types of transactions testing some acceptance criteria (i.e., mainly assertions) and calculating the average time to execute each transaction.

Transactions included different SQL statements to test intensive load in the database. The characteristics of each transaction are listed in Table 5.1.

| Transaction name | Workload type |
|---|---|
| Schedule Creation | Small 100% write workload |
| User Creation | Single write statement |
| Login | Small mixed workload |
| Logout | Single write statement similar to User Creation |
| Failed Login | Small mixed workload having only the last operation as write |
| Schedule Modification | Big mixed workload having a write transaction from the beggining |
| Schedule Retrieval | Small 100% read workload |
| User Modification | Small mixed workload similar to Login |
| Schedule Deletion | Small 100% write workload |
| User Deletion | Single write statement |
| State Report | Big 100% read workload |
| Auditing Report | Single read statement |

Table 5.1: Characteristics of the transactions executed during SteelDB evaluation.

We list below the statements contained in each transaction:

**Schedule Creation**

```
insert into TIMETABLE (NAME, OPERATIONAL_AREA, VERSION, TIMETABLEUID)
values (?, ?, ?, ?)

insert into PERIOD (END, START, TIMETABLE, VERSION, PERIODUID) values
(?, ?, ?, ?, ?)

--twice:
insert into CONTROL (MODE, OFFSET, RANK, PERIOD, SPECIAL_DAY_SERVICE,
TARGET_STATE, TIME, VERSION, CONTROLUID) values (?, ?, ?, ?, ?, ?, ?,
?, ?)
```

**User Creation**

```
insert into USER (CLIENT, DEATH_DATE, DELETED, EMAIL, FAILED_LOGINS,
LOCKED, LOCKED_DATE, LOGIN, NAME, OPERATIONAL_AREA, PASSWORD, SALT,
VERSION, USERUID) values (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
```

**Login**

```
select U.USERUID, U.CLIENT, U.DEATH_DATE, U.DELETED, U.EMAIL,
U.FAILED_LOGINS, U.LOCKED, U.LOCKED_DATE, U.LOGIN, U.NAME,
U.OPERATIONAL_AREA, U.PASSWORD, U.SALT, U.VERSION from USER U where
lower(U.LOGIN)=? and U.DELETED=? limit ?

insert into AUDIT_ACTION (CONTEXT, ACTION_DATE, LOGIN, TEXT, TYPE,
USERUID, VERSION, AUDIT_ACTIONUID) values (?, ?, ?, ?, ?, ?, ?, ?)
```

**Logout**

```
insert into AUDIT_ACTION (CONTEXT, ACTION_DATE, LOGIN, TEXT, TYPE,
USERUID, VERSION, AUDIT_ACTIONUID) values (?, ?, ?, ?, ?, ?, ?, ?)
```

**Failed Login**

```
select U.USERUID, U.CLIENT, U.DEATH_DATE, U.DELETED, U.EMAIL,
U.FAILED_LOGINS, U.LOCKED, U.LOCKED_DATE, U.LOGIN, U.NAME,
U.OPERATIONAL_AREA, U.PASSWORD, U.SALT, U.VERSION from USER U where
lower(U.LOGIN)=? and U.DELETED=? limit ?
```

```
select A.PARAMETERUID, A.NAME, A.VALUE, A.VERSION from
APPLICATION_SETTING A where A.NAME=? limit ?
```

```
update USER set CLIENT=?, DEATH_DATE=?, DELETED=?, EMAIL=?,
FAILED_LOGINS=?, LOCKED=?, LOCKED_DATE=?, LOGIN=?, NAME=?,
OPERATIONAL_AREA=?, PASSWORD=?, SALT=?, VERSION=? where USERUID=?
and VERSION=?
```

**Schedule Modification**

```
select TT.TIMETABLEUID, TT.NAME, TT.OPERATIONAL_AREA, TT.VERSION from
TIMETABLE TT where TT.TIMETABLEUID=?
```

```
update TIMETABLE set NAME=?, OPERATIONAL_AREA=?, VERSION=? where
TIMETABLEUID=? and VERSION=?
```

```
select P.PROFILEUID, P.DESCRIPTION, P.NAME, P.OPERATIONAL_AREA,
P.TIMETABLE, P.VERSION from PROFILE P where P.TIMETABLE=?
```

```
select S.SERVICEUID, S.NAME, S.PROFILE, S.TIMETABLE, S.VERSION from
SERVICE S where S.TIMETABLE=?
```

```
select D.DTCUID, D.BIRTH_DATE, D.CLIENTUID, D.DEATH_DATE, D.DELETED,
D.DISTRICTUID, D.HASPHOTOCELL, D.IPADDRESS, D.LATITUDE, D.LONGITUDE,
D.MUNICIPALITYUID, D.NAME, D.OPERATIONAL_AREAUID, D.PROFILEUID,
D.RTUUID, D.SECSUBSTATION, D.TIMETABLEUID, D.VERSION from DTC D where
D.TIMETABLEUID=? and D.DELETED=?
```

```
select P.PERIODUID, P.END, P.START, P.TIMETABLE, P.VERSION from
PERIOD P where P.PERIODUID=?
```

```
update PERIOD set END=?, START=?, TIMETABLE=?, VERSION=? where
PERIODUID=? and VERSION=?
```

```
select C.CONTROLUID, C.MODE, C.OFFSET, C.RANK, C.PERIOD,
C.SPECIAL_DAY_SERVICE, C.TARGET_STATE, C.TIME, C.VERSION from CONTROL
C where C.CONTROLUID=?
```

```
update CONTROL set MODE=?, OFFSET=?, RANK=?, PERIOD=?,
SPECIAL_DAY_SERVICE=?, TARGET_STATE=?, TIME=?, VERSION=? where
CONTROLUID=? and VERSION=?
```

```
select C.CONTROLUID, C.MODE, C.OFFSET, C.RANK, C.PERIOD,
C.SPECIAL_DAY_SERVICE, C.TARGET_STATE, C.TIME, C.VERSION from CONTROL
C where C.CONTROLUID=?
```

```
update CONTROL set MODE=?, OFFSET=?, RANK=?, PERIOD=?,
SPECIAL_DAY_SERVICE=?, TARGET_STATE=?, TIME=?, VERSION=? where
CONTROLUID=? and VERSION=?
```

**Schedule Retrieval**

```
select TT.TIMETABLEUID, TT.NAME, TT.OPERATIONAL_AREA, TT.VERSION from
TIMETABLE TT where TT.TIMETABLEUID=?

select P.PERIODUID, P.END, P.START, P.TIMETABLE, P.VERSION from
PERIOD P where P.TIMETABLE=?

select C.CONTROLUID, C.MODE, C.OFFSET, C.RANK, C.PERIOD,
C.SPECIAL_DAY_SERVICE, C.TARGET_STATE, C.TIME, C.VERSION from CONTROL
C where C.PERIOD=?
```

**User Modification**

```
select U.USERUID, U.CLIENT, U.DEATH_DATE, U.DELETED, U.EMAIL,
U.FAILED_LOGINS, U.LOCKED, U.LOCKED_DATE, U.LOGIN, U.NAME,
U.OPERATIONAL_AREA, U.PASSWORD, U.SALT, U.VERSION from USER U where
U.USERUID=?

update USER set CLIENT=?, DEATH_DATE=?, DELETED=?, EMAIL=?,
FAILED_LOGINS=?, LOCKED=?, LOCKED_DATE=?, LOGIN=?, NAME=?,
OPERATIONAL_AREA=?, PASSWORD=?, SALT=?, VERSION=? where USERUID=? and
VERSION=?
```

**Schedule Deletion**

```
delete from CONTROL where CONTROLUID=? and VERSION=?

delete from PERIOD where PERIODUID=? and VERSION=?

delete from TIMETABLE where TIMETABLEUID=? and VERSION=?
```

**User Deletion**

```
delete from USER where USERUID=? and VERSION=?
```

**State Report**

```
select SS_.SERVICE_TIMERUID, SS_.BEGIN_DATE, SS_.CLIENTEUID,
SS_.DISTRICTUID, SS_.DTCUID, SS_.DTC_NAME, SS_.END_DATE, SS_.MINUTES,
SS_.MUNICIPALITYUID, SS_.OPERATIONAL_AREAUID, SS_.POWER, SS_.RUNNING,
SS_.SERVICEUID, SS_.SERVICE_NAME, SS_.STATE, SS_.VERSION from
SERVICE_STOPWATCH SS_ where SS_.RUNNING=? limit ?

-- x10:
select DS.DTC_SERVICEUID, DS.BIRTH_DATE, DS.DEATH_DATE, DS.DELETED,
DS.DTCUID, DS.OPMODE, DS.NAME, DS.STATE, DS.VERSION from DTC_SERVICE
DS where DS.DTC_SERVICEUID=?
```

**Auditing Report**

```
select AA.AUDIT_ACTIONUID, AA.CONTEXT, AA.ACTION_DATE, AA.LOGIN,
AA.TEXT, AA.TYPE, AA.USERUID, AA.VERSION  from AUDIT_ACTION AA
limit ?
```

The *first scenario*, for functional evaluation, tested if the system could tolerate the expected faults. Requests were performed from the two SLS application servers executing queries in SteelDB in three phases, with 5, 10 and 20 simultaneous sessions each.

The tests were executed in three steps, containing different faulty behaviors:

**Step 1:** A non-master SteelDB replica is offline.

**Step 2:** A non-master SteelDB replica is compromised, returning inconsistent results.

**Step 3:** The execution starts with the four SteelDB replicas online and correct but, during the execution of tests one of the replicas is disconnected.

The *second scenario*, for performance evaluation, executed similar operations as the first, but the goal of that scenario was to measure the performance of the components in different architecture configurations.

As in the *first scenario*, the tests performed simulated two application servers in three phases having 5, 10 and 20 client sessions.

The tests were executed in four steps, being the first without the presence of faults:

**Step 1:** All four SteelDB replicas are online and correct during the entire test execution.

**Step 2:** A non master SteelDB replica is offline.

**Step 3:** A non master SteelDB replica is compromised, returning inconsistent results.

**Step 4:** The execution starts with the four SteelDB replicas online and correct but, during the execution of tests one of the replicas is disconnected.

The different configurations tested during the performance tests were:

- Baseline @TC: Direct connection from two application servers in the TrustedInfrastructure Cloud to one DB server in the Trusted Infrastructure;
- Baseline @EC2: Direct connection from two application servers in the TrustedInfrastructure Cloud to one DB server at Amazon EC2;
- Baseline @Azure: Direct connection from two application servers in the TrustedInfrastructure Cloud to one DB server in Windows Azure;
- SteelDB @CoC: Two application servers and two SteelDB replicas in the TrustedInfrastructure Cloud and two replicas in public clouds;
- SteelDB @TC: Two application servers and four SteelDB replicas in the TrustedInfrastructure Cloud.

While the two evaluation scenarios tested SteelDB execution with and without the presence of faults, they did not include tests to validate fault recovery.

To validate fault recovery we created a script to execute database operations and simulate Byzantine and crash scenarios during the execution. Those tests validated the leader change and state transfer protocols provided by Dura-SMaRt and the use of those protocols by SteelDB. It also validated SteelDB protocols to change master and transfer state, using database dumps as the checkpoint plus operations logs.

## 5.3 Results

After setup the environment for the tests execution, we deployed all the components to the virtual machines created in the TrustedInfrastructure Cloud and the public clouds.

One of the integration partners provided a script to create several threads, simulating concurrent execution and collected result data to be processed later. Every thread executed several times each transaction (10 for login and logout transactions and 100 for the others) described in table 5.1 and listed in the previous section.

We executed the scripts several times for each step and each number of clients simulated. When counting the number of steps, clients and architecture configurations tested we performed more than 200 script executions.

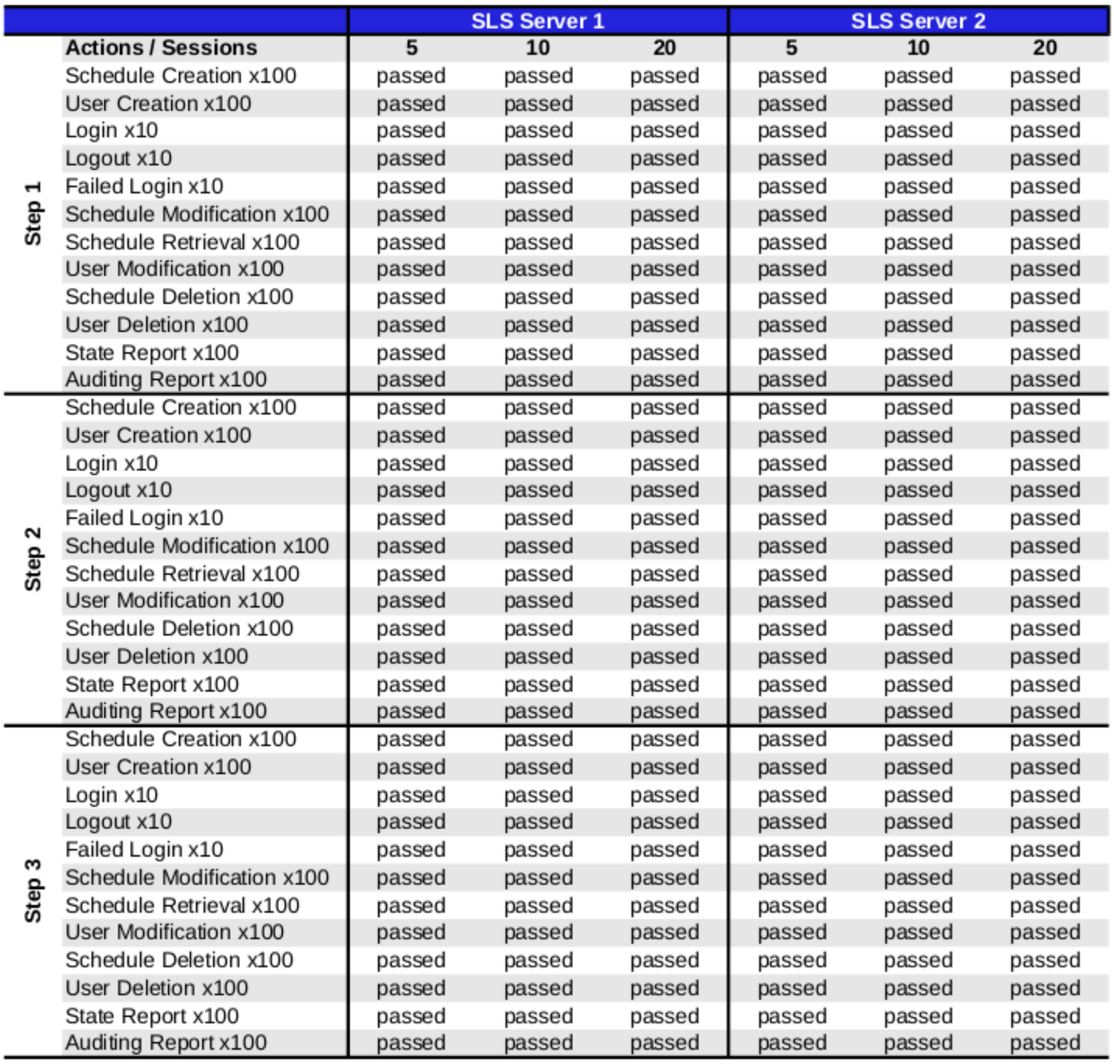
### Results for the Functional Tests

As the tests imposed a huge load on both SteelDB and Dura-SMaRt protocols, including multiple client sessions and concurrency, we discovered bugs and limitations in the protocol, like the FIFO order problem described in Section 4.3.1. That allowed us to refine the protocols and have a robust version tolerating faults and working correctly according to the tests.

Table 5.2 shows that all tests passed having the requests being performed from the two SLS application servers for all transaction types in the steps defined.

### Results for the Performance Tests

The tests for the performance evaluation of SteelDB executed several transactions in different architecture configurations. The script used to run the queries measured the average time to execute



| | | SLS Server 1 | | | SLS Server 2 | | |
|---|---|---|---|---|---|---|---|
| | Actions / Sessions | 5 | 10 | 20 | 5 | 10 | 20 |
| Step 1 | Schedule Creation x100 | passed | passed | passed | passed | passed | passed |
| | User Creation x100 | passed | passed | passed | passed | passed | passed |
| | Login x10 | passed | passed | passed | passed | passed | passed |
| | Logout x10 | passed | passed | passed | passed | passed | passed |
| | Failed Login x10 | passed | passed | passed | passed | passed | passed |
| | Schedule Modification x100 | passed | passed | passed | passed | passed | passed |
| | Schedule Retrieval x100 | passed | passed | passed | passed | passed | passed |
| | User Modification x100 | passed | passed | passed | passed | passed | passed |
| | Schedule Deletion x100 | passed | passed | passed | passed | passed | passed |
| | User Deletion x100 | passed | passed | passed | passed | passed | passed |
| | State Report x100 | passed | passed | passed | passed | passed | passed |
| | Auditing Report x100 | passed | passed | passed | passed | passed | passed |
| Step 2 | Schedule Creation x100 | passed | passed | passed | passed | passed | passed |
| | User Creation x100 | passed | passed | passed | passed | passed | passed |
| | Login x10 | passed | passed | passed | passed | passed | passed |
| | Logout x10 | passed | passed | passed | passed | passed | passed |
| | Failed Login x10 | passed | passed | passed | passed | passed | passed |
| | Schedule Modification x100 | passed | passed | passed | passed | passed | passed |
| | Schedule Retrieval x100 | passed | passed | passed | passed | passed | passed |
| | User Modification x100 | passed | passed | passed | passed | passed | passed |
| | Schedule Deletion x100 | passed | passed | passed | passed | passed | passed |
| | User Deletion x100 | passed | passed | passed | passed | passed | passed |
| | State Report x100 | passed | passed | passed | passed | passed | passed |
| | Auditing Report x100 | passed | passed | passed | passed | passed | passed |
| Step 3 | Schedule Creation x100 | passed | passed | passed | passed | passed | passed |
| | User Creation x100 | passed | passed | passed | passed | passed | passed |
| | Login x10 | passed | passed | passed | passed | passed | passed |
| | Logout x10 | passed | passed | passed | passed | passed | passed |
| | Failed Login x10 | passed | passed | passed | passed | passed | passed |
| | Schedule Modification x100 | passed | passed | passed | passed | passed | passed |
| | Schedule Retrieval x100 | passed | passed | passed | passed | passed | passed |
| | User Modification x100 | passed | passed | passed | passed | passed | passed |
| | Schedule Deletion x100 | passed | passed | passed | passed | passed | passed |
| | User Deletion x100 | passed | passed | passed | passed | passed | passed |
| | State Report x100 | passed | passed | passed | passed | passed | passed |
| | Auditing Report x100 | passed | passed | passed | passed | passed | passed |

Table 5.2: Results for functional tests.

the transactions. We merged the results obtained from the two application servers from which the tests were executed to have a final transaction time average for each transaction type. We display the results in Figure 5.2

In the top box, with the results for the execution of transactions by 20 simultaneous clients from each application server (40 clients total), it is possible to notice that the TC configuration has worst results than CoC. Recall that the TC configuration includes the four SteelDB replicas,

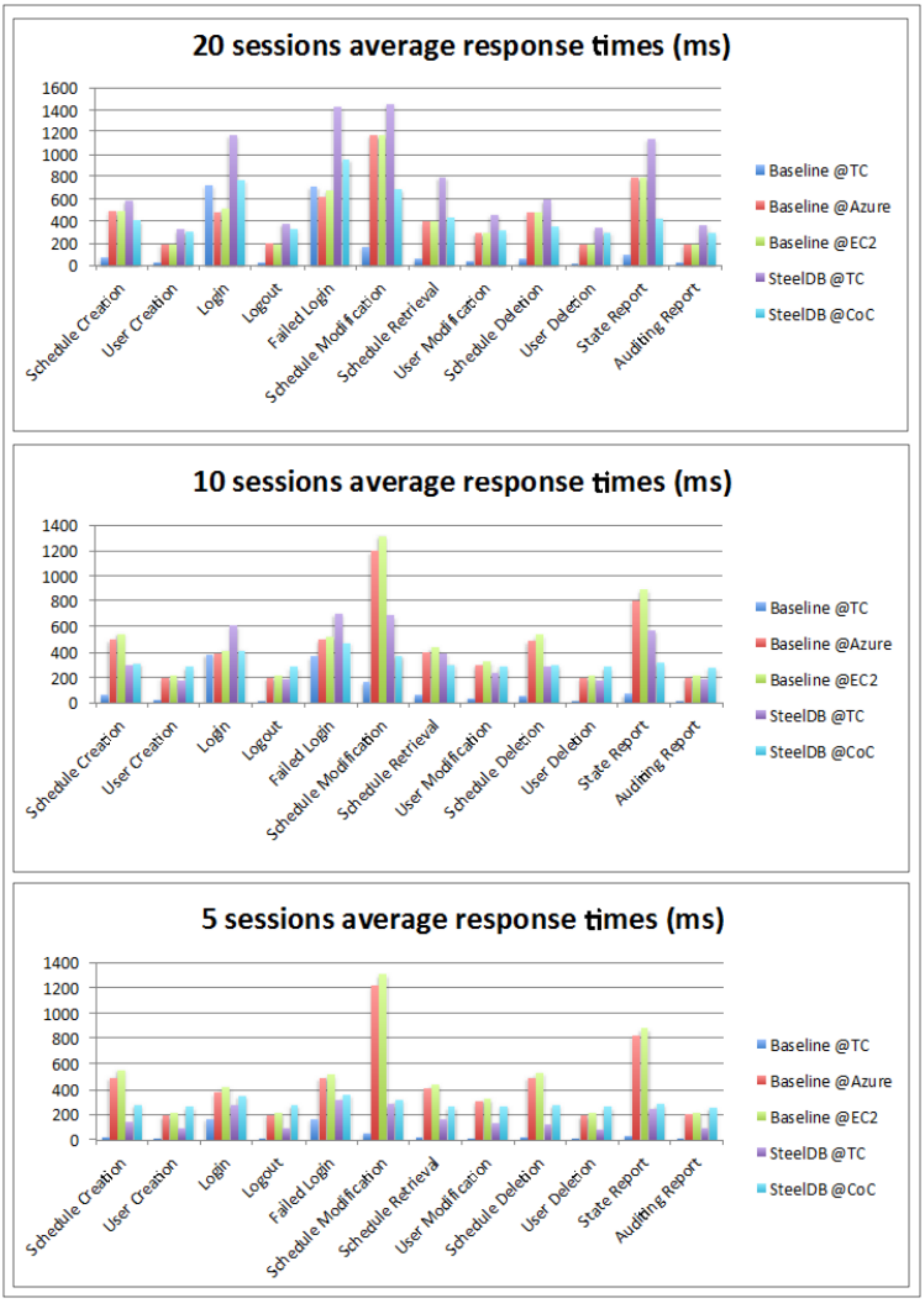


Figure 5.2: Results for the performance tests - Step 1.

while the CoC includes two replicas in the TC and two in the public clouds. In principle, this is a surprising result, as having four VMs in the same network should yield better results than having four VMs exchanging messages over a wide area network. This makes clear that the performance in the TC provided by the partner is affected by having six VMs being executed in a single physical machine. To minimize this problem we will analyse the following results based on the configuration with 5 clients in each application server, which is less influenced by the hardware limitation.

The first thing to notice when comparing the times to execute the different types of transaction is that configurations with clients in TC invoking queries directly to the database servers hosted in the public clouds has the worst results. This is true for both read and write-dominated workloads. Schedule Modification, for instance, takes 1306 milliseconds, on average, to be executed in EC2, but takes 73% less time when executed with SteelDB in CoC (318 ms). This is explained by the optimization made in SteelDB to execute all the statements first in the master replica and replicate the data only by commit time. That transaction execution would invoke only one wide area network (WAN) messages, plus the messages from the ordering protocol of BFT-SMARt, against eleven without SteelDB. The gain in performance is also true for the State Report transaction which takes 881 ms in EC2 against 293 ms in CoC, 67% less. This is also explained by the optimization where we read operations from only the master as we assume the component to be trusted.

For a transaction with only one statement, the performance is not much different from direct invoking operations in the cloud, though. User deletion, which has only one statement, takes 213 ms to be executed in EC2 and 265 ms to be executed in CoC, 24% more. This can be explained by the fact that despite of both cases perform exchange messages for two queries over the WAN, additional messages are exchanged between the replicas, increasing the total time to execute the transaction. But executing a transaction with three operations already has better results with SteelDB. Schedule Deletion takes 527 ms to be executed from the server in TC to EC2, taking 276 ms in CoC with SteelDB, 48% less.

Figure 5.3 displays the ratio between the execution time of transactions in different architectures. First, it is necessary to say that although executing transactions directly from the SLS servers to the database server in TC has the fastest responses, but tolerates no faults at all.

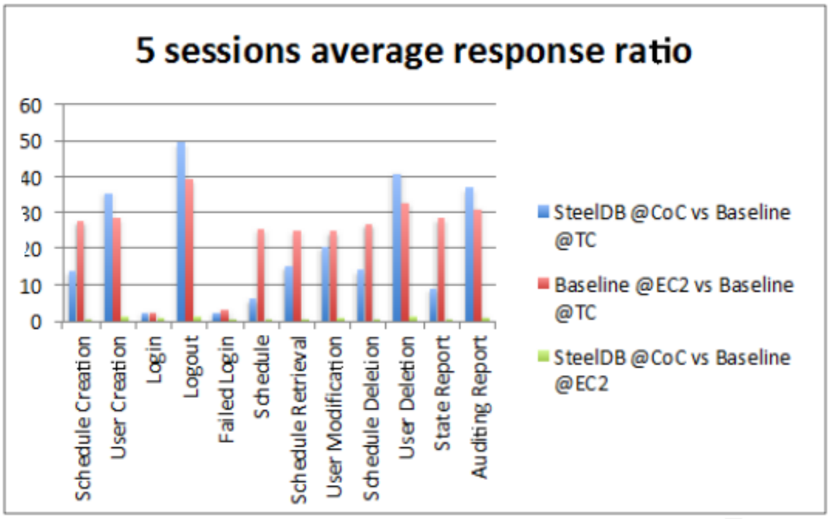


Figure 5.3: Ratio between executing transactions in different architecture configurations.

Comparing results from SteelDB execution in CoC with baseline executions at TC shows that the baseline at TC performs much faster than SteelDB, as no replication or wide area communication is needed. Small transactions like User Creation takes 8 ms to be performed in TC while takes 266 ms in CoC, 33 times more. With bigger transactions, like Schedule Modification, the difference is not that big, although considerable. It takes 51 ms in TC versus 318 with SteelDB in CoC, about 6 times more.

Comparing the same scenarios for executions with 20 simultaneous sessions we can notice a small increase in time for both baseline execution in TC and SteelDB in CoC, for both types of transactions. Baseline execution of User Creation in TC takes 25 ms, while replication through SteelDB in CoC takes 308 ms, 12 times more. Baseline execution of Schedule Modification in TC takes 167 ms versus 688 with SteelDB in CoC, 4 times more.

When comparing SteelDB in CoC with directly connection between the application servers in TC to a database server in EC2 the difference is much smaller, showing the cost of wide area communication.

Steps 2, 3 and 4 of the performance tests evaluated the execution of transactions over SteelDB in the presence of different types of faults. A comparison between results from Step 1, the fault free scenario with results for step 2, in which one replica is offline during the whole execution of the test are displayed in Figure 5.4. It is possible to notice that the time to execute transactions in SteelDB in the presence of faults is never greater than 5% of the time in the execution without faults. In some cases the time is even smaller. This can be explained by the fact that the normal replication operation of BFT-SMaRt expects $n - f$ replicas to reply before returning the results to the user. That always discards the replies from the $f$ slowest replicas. As displayed in this

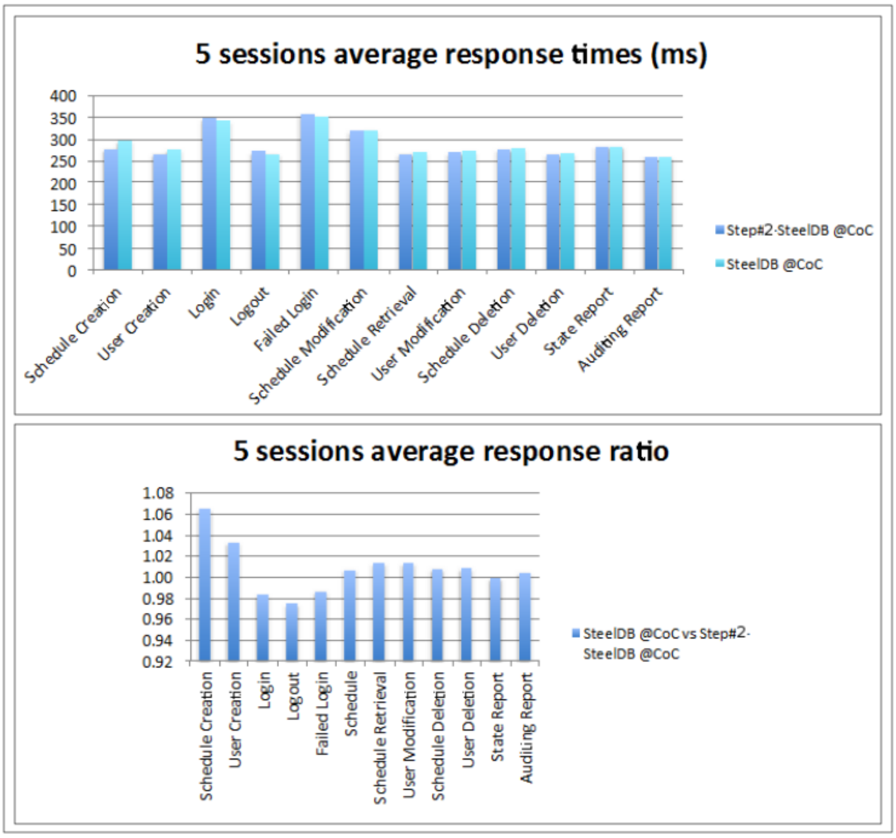


Figure 5.4: Performance of SteelDB in the presence of failures.

section, the response times for Microsoft Azure and Amazon EC2 are pretty close, which could explain the 5% difference in the results. We omit here the comparison between results from step 1 with steps 3 and 4, since the results are similar to step 2.

### TClouds final demo

During the TClouds project final demo we presented the SLS execution over SteelDB with BFT-SMaRt deployed in the Cloud-of-Clouds architecture. We simulated server crash and recovery without service disruption, to present the state transfer mechanism of BFT-SMaRt. We also displayed the tolerance of Byzantine faults simulating a compromised replica. Finally we injected a BFT-SMaRt leader change and SteelDB master change in front of a panel of project reviewers. During the whole presentation the system behaved as expected, having the SLS application displayed constant execution without errors.

## 5.4 Final Remarks

The evaluation of SteelDB during the integration tests in the last phase of the TClouds project was very intensive. Several use cases where defined to attest fault tolerance during the concurrent use of the middleware. During the tests execution we found problems like the requirement for FIFO ordering of messages by BFT-SMaRt described in section 4.3.1. The tests helped us to identify the problems and have a more resilient version of the protocol.

Tests for functional execution of the middleware proved that it can tolerate different types of faults provide resilient responses to the application. Tests for performance evaluation showed that, for some transaction types, SteelDB can present better response times for replication over a cloud-of-clouds than execution of the client application connecting to a database server over a wide area network. When comparing the performance of SteelDB over CoC with its execution replicated over a local area network, the local area execution presents always faster response times, although it does not tolerate co-related faults like a power outage in the whole data center. Finally, when comparing database access over SteelDB replicated in a CoC with directly access to the database in the same network of the client, the results are much faster in the local execution. Tests were executed from 6 to 40 times faster in local transactions, but did not tolerate any fault.

# Chapter 6

# Related Work

In this section we describe the work related to state management and transfer in consensus and state machine replication frameworks. We also mention the work performed on database management systems replication middleware.

## 6.1 Durability on State Machine Replication

Over the years, there has been a reasonable amount of work about stable state management in main memory databases (see [26] for an early survey). Parallel logging (see Section 2.3.1) shares some ideas with classical techniques such as group commit and pre-committed transactions [22] and the creation of checkpoints in background has also been suggested [41]. Our techniques were however developed with the SMR model in mind, and therefore, they leverage the specific characteristics of these systems (e.g., log groups of requests while they are executed, and schedule checkpoints preserving the agreement quorums).

Durability management is a key aspect of practical crash-FT SMR-like systems [10, 16, 32, 35, 52, 60]. In particular, making the system use the disk efficiently usually requires several hacks and tricks (e.g., non-transparent copy-on-write, request throttling) on an otherwise small and simple protocol and service specification [16]. These systems usually resort to dedicated disks for logging, employ mostly synchronized checkpoints and fetch the state from a leader [16, 32, 52]. A few systems also delay state transfer during load-intensive periods to avoid a noticeable service degradation [32, 60]. All these approaches either hurt the SMR elegant programming model or lead to the problems described in Section 2.2, like latency on service response due to log writing to disk and even stop on service response for several seconds while a checkpoint of the service state is being taken.

For instance, recent consistent storage systems such as Windows Azure Storage [12] and Spanner [20] use Paxos together with several extensions for ensuring durability. We believe works like ours can improve the modularity of future systems requiring durable SMR techniques.

BFT SMR systems use logging, checkpoints, and state transfer, but the associated performance penalties often do not appear in the papers because the state is very small (e.g., a counter) or the checkpoint period is too large (e.g., [13, 18, 29, 36, 40, 59, 58]). A notable exception is UpRight [17], which implements durable state machine replication, albeit without focusing on the efficiency of logging, checkpoints and state transfer. UpRight offers different strategies for taking checkpoints from the service state. It implements a hybrid checkpoint mechanism where checkpoint is taken from the service existing code and deltas are generated on predefined periods. It offers three strategies to generate the deltas: stop and copy, helper processes where it takes a copy of the data in memory where it applies the changes during the checkpoint generation and copy-on-write. UpRight also allows the application to implement custom checkpointing strategies. UpRight strategies to reduce the impact that checkpoint generation has on the service execution requires that checkpoints are taken frequently. Tests described in the paper mentioned checkpoints being taken on every 50 requests. If one wants to sustain a high-throughput (as reported in the papers) for non-trivial states, the use of our techniques is fundamental. Moreover, any implementation of proactive recovery [13, 56, 51] requires an efficient state transfer to make rejuvenated replicas up to date.

The work in [16] describes the several challenges faced to implement a Paxos consensus algorithm to be used by the Chubby [11] coordination service. Authors mentioned that during the implementation of the protocol they found several problems not addressed by the algorithm. The log of operations decided by a consensus instance includes information view numbers, leader and client ids, epoch numbers and leases. It was necessary to include these informations in the checkpoint together with the application state to make it possible to restore the application state plus the consensus instance when a checkpoint is installed.

PBFT [13], uses a slightly different message ordering protocol than BFT-SMaRt. Requests are sent first to a primary replica that proposes a sequence to backups through pre-prepare, prepare and commit message exchange rounds before providing the request to be executed to the service. During all phases of the protocol the decisions are logged and only after a checkpoint is validated the log is discarded. Despite the differences in the ordering protocol, PBFT was one of the few works that explicitly deal with the problem of optimizing checkpoints and state transfer. The proposed mechanism was based on copy-on-write and delta-checkpoints. It can store several checkpoints and only after a checkpoint is confirmed to be valid through an exchange of messages containing checkpoint digests it can be used in a state transfer. This mechanism is complementary to our techniques, as we could use it together with sequential checkpoints and also to fetch checkpoint pages in parallel from different replicas to improve the state transfer. However, the use of copy-on-write may require the service definition to follow certain abstractions [14, 17], which can increase the complexity of the programming model. Additionally, this mechanism, which is referred in many subsequent works (e.g., [29, 40]), only alleviates but does not solve the problems discussed in Section 2.2. Also, the work [13] does not include an evaluation of the effect of state manage and transfer in the execution of the protocol.

A few works have described solutions for fetching different portions of a database state from several "donors" for fast replica recovery or database cluster reconfiguration (e.g., [38]). The same kind of techniques was employed for fast replica recovery in group communication systems [37] and, more recently, in main-memory-based storage [48]. There are three differences between these works and ours. First, these systems try to improve the recovery time of faulty replicas, while CST main objective is to minimize the effect of replica recovery on the system performance. Second, we are concerned with the interplay between logging and checkpoints, which is fundamental in SMR, while these works are more concerned with state snapshots. Our work has a broader scope in the sense that it includes a set of complementary techniques for Byzantine and crash faults in SMR systems, while previous works address only crash faults.

## 6.2 Database Replication

Database management systems provide interfaces and tools to store and manipulate information. As information is a key component in business operations, mechanisms to provide availability and performance have been devised since decades ago. To provide scalability, availability and fault tolerance for database management systems, a common approach is use database replication. That requires dealing with several problems like network communication, concurrent access, data partitioning and state management.

The work in [15] describes several problems that had to be addressed when designing and implementing a database replication middleware. It shows use cases of fortune 500 companies where a DBMS outage of only one minute is unacceptable. Several large corporations have strong requirements where databases have to be available for 99,999% of the time. That means that a database can be offline for planned or unplanned maintenance for at most five minutes during a year. To achieve this level of service replication systems must address efficiently several problems in different layers:

**Multi-databse queries:** depending on the size of the service being managed, it is common to have queries and transactions that include different databases or schemas. This requires access management to databases and the middleware has to consider all databases when managing the database state.

**Isolation level:** 1-copy serializability [6] is a powerful technique to isolate transactions and prevent data corruption. The problem with it is that the constant use of locks on table or row levels reduces the concurrency access to data and decreases the performance of the system as operations may have to wait for locks to be released before being executed. Snapshot isolation [5] alleviates this problem but also brings other problems like the need of rollback transactions due to modified data.

**Heterogeneous clustering:** availability can be increased adding servers to clusters. As the number of clusters increases, problems like different computer architectures, database versions, operating systems and hardware configurations arises.

**Database internals:** several vendors design internal mechanisms to increase the performance of a DBMS. Temporary tables can be created during a transaction to manage temporary data. Sequences can be created to control the creation of unique numbers to identify objects.

**Determinism:** replicated systems has to execute deterministic operations to guarantee that a consistent state is common among different replicas. To achieve that, operations like getting the current time in a server or generate a random number have to be managed to produce the same value in all replicas of a system.

**SQL-level challenges:** databases can have internal tools, reserved words and even languages to provide additional services like stored procedures and efficient management of large amounts of data. This can make it difficult for the replication middleware to have access to database internals and replicate data that is manage by such tools.

**Failover and failback:** efficient mechanisms to detect that a replica failed and restore it are necessary to keep the system running and tolerate additional faults.

**State management:** state has to be checkpointed and logged efficiently, to be able to recover a faulty replica or include new replicas in the cluster, allowing the replication system to scale out.

Due to the number of challenges to be addressed to design and build a robust database replication middleware with acceptable availability and performance, some of the items described above sometimes are neglected or overlooked.

The work on [27] analyses software failures in database products. It analyses the differences between buying an off-the-shelf DBMS product and develop one, in terms of fault tolerance Off-the-shelf database management systems are usually cheaper than design and implement a customized solution for information management. Most of these systems offer replication to increase performance and tolerate faults. But buying and installing an off-the-shelf DBMS replication system may have some disadvantages like software bugs not yet detected, causing data corruption due to explicit or silent failures. Experiments performed on that work showed that bugs on both commercial and open-source products are not repeated when the database vendor and sometimes even the software version is changed. Several types of diversity like language, design, software and data can be implemented by having a replication middleware between the client and the database management system. The challenge in this case is that the design and implementation of a database replication middleware itself is quite complex and may require a huge time and effort to address questions like concurrent transaction processing between different vendors, SQL idioms incompatibility and state management between different vendors.

The work on [57] describes a Commit Barrier Scheduling scheme, named HRDB (Heterogeneous Replicated DB) to replicate databases and tolerate Byzantine faults. HRDB was designed to support diversity over database vendors. It has a commit barrier component that executes operations in a primary database before forward them to secondary databases. This logic requires that databases implement two-phase lock concurrency mechanisms. Some database vendors (Oracle and PostgreSQL, for instance) uses snapshot isolation [5] instead, so can not be used with HRDB.

HRDB assumes the ordering of statements decoupled from execution in a component called Shepherd. It is considered a trusted component, so Byzantine faults are expected to happen only in database replicas. It uses $3f + 1$ replicas to order and process the results, while it is necessary only $2f + 1$ replicas to execute the operations, meaning only $2f + 1$ databases. The Shepherd is composed by the agreement protocol along with a coordinator and one manager for each database replica. The manager can be a primary or a secondary, and the messages are executed first in the primary and then forwarded to the secondary managers, serializing the requests in the primary.

The recovery strategy used in HRDB assumes that replicas can fail in a crash or Byzantine way. When a replica is restarted the manager forwards it the operations to be processed. This mechanism requires the database to have a table to log the last operation executed to enable the manager to know if it the failure occurred before or after the commit of a transaction. The recovery protocol do not assume snapshots for database state. The database manager logs operations only before they are committed but if the database replica spent a long time offline, the log can be quite large.

In [25] it is defined a protocol to replicate database systems using state machine replication. This replication system, called Byzantium tolerates crash and Byzantine faults. It requires DBMS that provide snapshot isolation for concurrency management and messages delivered by the replication protocol in FIFO order. To optimize the protocol execution, Byzantium executes operations speculatively in one replica, called master and compares results from other replicas during commit time. It also has a multi-master version where clients can assume different masters, alleviating the load on a specific replica. This can also reduce latency when the client chooses a replica closer to it.

We used several ideas present in that work do define and implement SteelDB. Our work differs from Byzantium, as we do not assume FIFO ordering of requests from clients. We also do not implement the multi-master version. We focused though in different aspects like efficient state management, an aspect not mentioned in the Byzantium paper. We used BFT-SMART with the durability techniques presented in Chapter 2, without any further modification.

# Chapter 7

# Conclusion

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

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

The conception and implementation of a database replication middleware over the new durable version of BFT-SMART, which we called SteelDB, proved possible to implement complex services over the replication protocol without the need to perform internal changes in it. It was possible to manage complex database state - an area usually overlooked in database middleware replication protocols [15] - including database dump and session information for multiple clients using only BFT-SMART durable state management layer.

After the implementation of SteelDB, we evaluated it, together with BFT-SMART during the integration tests for the TClouds project. The tests exercised not only the fault-free execution cases but also included several scenarios simulating crash and Byzantine faults. It attested that our replication protocol is robust enough to be used by multiple clients concurrently.

In a configuration with replicas distributed over a hybrid architecture including a private and a public clouds we proved possible to add durability to database management systems without the need for change in client code nor DBMS internals. Although such configuration provides tolerance for catastrophic failures, replication over wide area networks impose a considerable drop in the middleware performance.

## 7.1 Future Work

Although replicas can be added to a BFT-SMART system to increase fault tolerance, this does not provide scalability, as every replica has to execute all update requests due to the determinism requirement of state machine replication. A solution to that problem is to split the application state in several groups and use a SMR group to manage each portion [50]. Mechanisms to measure the size of the state and the need for new splits can be employed to provide the desired performance. Together with the mechanisms to split the state it is necessary to provide ways for clients to know where each portion of data is stored and perform requests in the correct SMR group or have a group to redirect requests to the correct group.

Improvements can also be made in SteelDB. During its development and evaluation we used databases from one vendor at a time (H2, MySQL and PostgreSQL). Although we designed a module to translate queries between different vendors we did not focus our efforts in implementing an efficient version for state management and transfer. Having a robust translation module implemented would make possible to have a version of the middleware providing diversity for vendor implementation. This would make possible to tolerate software bugs common to a vendor distribution [27].

# Bibliography

[1] Apache Tomcat. `http://tomcat.apache.org`.

[2] Hibernate ORM. `http://hibernate.org/orm/`.

[3] Project TCLOUDS – Trustworthy Clouds - Privacy and Resilience for Internet-scale Critical Infrastructure. `http://www.tclouds-project.eu/`, 2010.

[4] Alysson Bessani et. al. TClouds – Adaptive Cloud-of-Clouds Architecture, Services and Protocols. Deliverable D2.2.4, TClouds Consortium, September 2013.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10. ACM, 1995.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[7] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia. On the efficiency of durable state machine replication. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference (USENIX ATC 2013)*, pages 169–180, June 2013.

[8] A. Bessani, J. Souza, and E. Alchieri. State Machine Replication for the Masses with BFT-SMaRt. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN 2014)*, 2014.

[9] A. N. Bessani, E. P. Alchieri, M. Correia, and J. D. S. Fraga. Depspace: A byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM european conference on Computer Systems (EuroSys 2008)*, pages 163–176, 2008.

[10] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2011)*, 2011.

[11] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation (OSDI 2006)*, pages 335–350, 2006.

[12] B. Calder et al. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 33rd ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2011)*, pages 143–157. ACM, 2011.

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

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

[15] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: The gaps between theory and practice. In *Proceedings of the 2008 ACM SIGMOD international Conference on Management of Data*, 2008.

[16] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live - An engineering perspective. In *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2007)*, 2007.

[17] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riché. UpRight cluster services. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2009)*, pages 277–290, 2009.

[18] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems Design and Implementation (NSDI 2009)*, pages 153–168, 2009.

[19] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[20] J. Corbett et al. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI 2012)*, pages 251–264, 2012.

[21] J. Dean. Google: Designs, lessons and advice from building large distributed systems. In *Keynote at the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS 2009)*, Oct. 2009.

[22] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 1–8, 1984.

[23] J. Félix. Gestão de Estados Eficiente no Serviço de Coordenação Eficiente DDS. Mestrado em Informática (in Portuguese). Faculdade de Ciências da Universidade de Lisboa, 2012.

[24] D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI 2010)*, 2010.

[25] R. Garcia, R. Rodrigues, and N. Preguica. Efficient middleware for Byzantine fault-tolerant database replication. In *Proceedings of the 6th ACM european conference on Computer Systems (EuroSys 2011)*, pages 107–122, 2011.

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

[27] I. Gashi, P. Popov, and L. Strigini. Fault tolerance via diversity for off-the-shelf products: a study with sql database servers. *IEEE Transactions on Dependable and Secure Computing*, 4(4):280–294, 2007.

[28] J. Gray. The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB 1981)*, volume 7, pages 144–154. VLDB Endowment, 1981.

[29] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *Proceedings of the 5th ACM european conference on Computer Systems (EuroSys 2010)*, pages 363–376, 2010.

[30] V. Hadzilacos and S. Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical Report TR 94-1425, Department of Computer Science, Cornell University, May 1994.

[31] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS 1995)*, 1995.

[32] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for Internet-scale services. In *Proceedings of the 2010 USENIX conference on Annual Technical Conference (USENIX ATC 2010)*, 2010.

[33] Java Community Process. JSR-000221 JDBC API Specification. Available at `https://jcp.org/aboutJava/communityprocess/mrel/jsr221/index.html`, 2011.

[34] E. C. Julie, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *Proceedings of the 2004 USENIX conference on Annual Technical Conference (USENIX ATC 2004)*, pages 9–18, 2004.

[35] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 41st International Conference on Dependable Systems and Networks (DSN 2011)*, pages 245–256, 2011.

[36] R. Kapitza, B. Johannes, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys 2012)*, pages 295–308, 2012.

[37] R. Kapitza, T. Zeman, F. Hauck, and H. P. Reiser. Parallel state transfer in object replication systems. In *Proceedings of the 7th IFIP WG 6.1 international conference on Distributed applications and interoperable systems (DAIS 2007)*, pages 167–180, 2007.

[38] B. Kemme, A. Bartoli, and Özalp Babaoglu. Online reconfiguration in replicated databases based on group communication. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN 2001)*, pages 117–130, 2001.

[39] J. Kirsh and Y. Amir. Paxos for system builders: An overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS 2008)*, 2008.

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

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

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

[43] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.

[44] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the Harp file system. In *Proceedings of the 13th ACM SIGOPS Symposium on Operating Systems Principles (SOSP 1991)*, pages 226–238, 1991.

[45] J. Lorch, A. Adya, W. Bolosky, R. Chaiken, J. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. In *Proceedings of the 1st ACM european conference on Computer Systems (EuroSys 2006)*, pages 103–115, 2006.

[46] Marco Abitabile et. al. TClouds – Final Report on Evaluation Activities. Deliverable D3.3.4, TClouds Consortium, October 2013.

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

[48] D. Ongaro, S. M. Ruble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the 33rd ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2011)*, pages 29–41, 2011.

[49] Paulo Santos et. al. TClouds – Smart Lighting System Specification. Deliverable D3.2.1, TClouds Consortium, March 2011.

[50] F. Pedone, C. E. Bezerra, and R. van Renesse. Scalable state-machine replication. In *Proceedings of the 2014 International Conference on Dependable Systems and Networks (DSN 2014)*, 2014.

[51] M. Platania, D. Obenshain, T. Tantillo, R. Sharma, and Y. Amir. Towards a practical survivable intrusion tolerant replication system. Technical Report CNDS-2014-1, Department of Computer Science at Johns Hopkins University, Apr. 2014.

[52] J. Rao, E. J. Shenkita, and S. Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*, 4:243–254, Jan. 2011.

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

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

[55] J. Sousa and A. Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proceedings of the 9th European Dependable Computing Conference (EDCC 2012)*, pages 37–48, 2012.

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

[57] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in database systems using commit barrier scheduling. In *Proceedings of the 21st ACM SIGOPS symposium on Operating Systems Principles (SOSP 2007)*, pages 59–72, Oct. 2007.

[58] G. Veronese, M. Correia, A. Bessani, L. Lung, and P. Verissimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 62(1), 2013.

[59] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proceedings of the 28th IEEE International Symposium on Reliable Distributed System (SRDS 2009)*, pages 135–144, Sept. 2009.

[60] Y. Wang, L. Alvisi, and M. Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC 2012)*, 2012.