André Nunes Gomes Alves

Licenciado em Engenharia Informática

Healing replicas in a software component replication system

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : Nuno Manuel Ribeiro Preguiça, Prof. Auxiliar,
Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2013

Healing replicas in a software component replication system

I dedicate my dissertation to those I love.
To my father Aníbal, to my mother Fernanda, and to my grandmother Antónia.
A deep feeling of gratitude goes to them.

Acknowledgements

I wish to thank those that have been present since the beginning of the course, as without acknowledging them this work would not be complete.

I would like to express my deepest gratitude to my supervisor, Professor Nuno Preguiça, for the opportunity given, for caring, and for all the guidance given.

I would like to acknowledge my colleges for being there in the bad times, and for making good times happen. A special thanks goes to Fábio, Pedro, and João, for all the patience and caring shown, for all the time spent together, and for all the discussions that enlightened me.

I would like to express my deepest feelings of eternal gratitude towards my family, for creating all the conditions needed to complete my studies. To my father Aníbal, my mother Fernanda, and my grandmother Antónia, a special feeling of gratitude goes to all of you for always being there for me. I am, and forever will, be in debt with all of you, and I do not think that I will ever find a way to repay all the love given.

Abstract

Replication is a key technique for improving performance, availability and fault-tolerance of systems. Replicated systems exist in different settings – from large geo-replicated cloud systems, to replicated databases running in multi-core machines. One feature that it is often important is a mechanism to verify that replica contents continue in-sync, despite any problem that may occur – e.g. silent bugs that corrupt service state.

Traditional techniques for summarizing service state require that the internal service state is exactly the same after executing the same set of operation. However, for many applications this does not occur, especially if operations are allowed to execute in different orders or if different implementations are used in different replicas.

In this work we propose a new approach for summarizing and recovering the state of a replicated service. Our approach is based on a novel data structure, Scalable Counting Bloom Filter. This data structure combines the ideas in Counting Bloom Filters and Scalable Bloom Filters to create a Bloom Filter variant that allow both delete operation and the size of the structure to grow, thus adapting to size of any service state.

We propose an approach to use this data structure to summarize the state of a replicated service, while allowing concurrent operations to execute. We further propose a strategy to recover replicas in a replicated system and describe how to implement our proposed solution in two in-memory databases: H2 and HSQL. The results of evaluation show that our approach can compute the same summary when executing the same set of operation in both databases, thus allowing our solution to be used in diverse replication scenarios. Results also show that additional work on performance optimization is necessary to make our solution practical.

Keywords: component replication, fault-tolerance, performance, multi-core processor

Resumo

A replicação é uma técnica fundamental para o melhoramento da performance, disponibilidade e tolerância a faltas de sistemas. Os sistemas replicados variam na sua implementação – desde sistemas geo-replicados na *cloud*, a bases de dados replicadas correndo em máquina multi-core. Uma característica que muitas vezes é importante possuírem é um mecanismo para verificar que o conteúdo de uma réplica continua sincronizado, mesmo na presença de faltas.

As técnicas tradicionais para sumarizar o estado de um serviço requerem que o seu estado interno seja exatamente o mesmo após este ter executado o mesmo bloco de operações. Porém para muitas aplicações tal não acontece, particularmente no caso em que as operações pode ser executadas em diferentes ordens, ou quando replicas correm diferentes implementações.

Neste trabalho é proposta uma nova abordagem para sumarizar e recuperar o estado de um serviço replicado. Esta abordagem baseia-se numa nova estrutura de dados, Scalable Counting Bloom Filter que combina as ideas presentes nos Counting Bloom Filters e nos Scalable Bloom Filters, suportando operações de remoção e capacidade de escalar conforme as necessidades do serviço. Propõe-se o uso da estrutura para sumarizar o estado de um serviço replicado e uma estratégia para a recuperação de replicas para serviços replicados, sendo descrita a forma como esta solução é implementada em duas bases de dados: H2 e HSQL. Os resultados da avaliação mostram que esta abordagem é capaz de gerar o mesmo sumário de estado quando são executados os mesmos blocos de operações, permitindo que esta solução seja usada em diversos cenários de replicação. Contudo, a solução requer optimizações para que seja aplicável.

Palavras-chave: replicação de componentes, tolerância a faltas, desempenho, processador multi-core

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context

Moore's law states that transistor density on integrated circuits doubles about every two years [1, 2]. Until relatively recently, this implied a relation between the amount of transistors in a chip and the processor's performance. That's because the additional transistors were mainly used to boost processor frequency and increase fast local memory. All this evolution was transparent to the software and the faster the hardware, the faster the software would run.

However, beginning around the middle of last decade, increasing clock frequency started to hit a wall. As a result, as it was still possible to increase the number of transistors in a chip, chipmakers start producing processors with multiple cores [3]. This changed the computational paradigm. From the perspective of the application these were no longer seen as a single processor, but instead as complete independent processors.

The problem is that applications created with a decades old design — i.e., including a single thread to execute a single thread of instructions — don't benefit from additional cores to do more work independently. Thus, these applications are unable to improve performance solely on the basis of running on a multi-core processor. Ir order to exploit the performance of these multi-core processors, reducing the time it takes applications to accomplish a given task, the work has to be split into pieces, so that the processors can work on them in parallel (multi-threading). This implies that the application needs to be rewritten, and its work needs to be split into pieces suitable for multi-threading.

1.2 Motivation

In distributed systems, replication has been used to improve performance of systems. RepComp tries to explore a similar replication approach to improve the performance of applications running in multi-core systems without requiring application to be rewritten. The intuition is that in single threaded applications, by keeping multiple replicas of a given component, it might be possible to use the implementation with best performance to run each operation more efficiently. In applications that have multiple threads, relying on multiple replicas can improve performance by decreasing the contention among multiple threads, as concurrent operations would run without any interaction in different replicas.

An alternative to multi-threading is to use state machine replication, more specifically component replication [4], treating multi-core processors as distributed systems. Thus allowing for single-threaded applications to exploit multi-core processor performance without being rewritten.

Although component replication allows for applications to exploit the benefits of running on multi-core processors, it also introduces a problem that needs addressing.

The problem with this technique is that in order for applications to be able to have improved throughput, with correct semantics, it is necessary to keep replicas synchronized. To this end, the system must include a replication algorithm to control the access to the replicas and their update.

One important issue that arises is to guarantee that all replicas are kept consistent. The replication algorithm must guarantee this property. In some cases, the execution of operations in a given replica may be slow and the replica become stale (when compared to others). In the presence of bugs, the state of a replica may become incorrect. To guarantee that the system continues working correctly, it is necessary to detect and repair this situation. This work focus on these problems in the context of RepComp project.

1.3 Implemented Solution

The RepComp project aims at exploring software component replication to improve performance and fault-tolerance of applications running in multi-core machines. The main idea is to replace software components used by the application with macro-components that include several replica of the same component specification — each replica may have the same or a different implementation of the same interface (e.g., a set macro-component may be composed by a replica implemented with a tree and other replica implemented with a hashtable).

MacroDB [5], an example of a Macro-Component [4], is a system aimed to improve *DBMS*s - specifically in-memory *DBMS*s - performance on multi-core processor platforms.

MacroDB is built as a *middleware* intended to be a transparent layer between clients

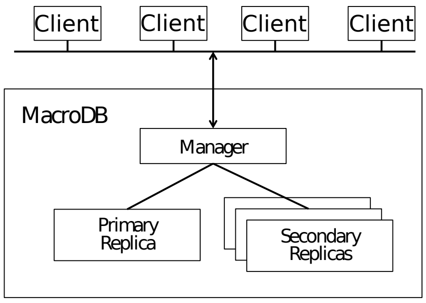


Figure 1.1: MacroDB Architecture (Figure taken from [5])

and the underlaying *DBMS*s. It is based on the master/slave architecture (Figure 1.1). All replicas have a copy of the master's database. The master node is responsible for all coordination efforts, being in charge of processing all update transactions and propagating any modifications to the database — after committing them — to all the slave nodes, shielding slave replicas from being affected by an aborted transaction on its side. Therefore ensuring that they will not go into an inconsistent state when a client aborts a transaction.

Contrary to what happens with update transactions, read transactions always run at one of the slaves as a way to improve the throughput of the system. Running read transactions in the slave replicas reduces contention among update and read-only transactions, thus improving the performance of read-only and update transactions. Yet before routing a read operation to a slave, the system has to guarantee that the slave is up-to-date. This is done by having a *counter* in each replica that is incremented when a commit occurs. So, when a read transaction arrives, only replicas that have the same counter value are eligible to run the operation. Electing a suitable candidate can go even further, in addition to the *counter* having to correspond in value with the master's *counter*, it may also be applied one of three election algorithms:

- *Round-Robin*, replicas form a circular queue and for each election we cycle through the queue;
- Directly assigning a slave replica to be used for a given client;
- Selecting the most lightly loaded replica.

Although MacroDB promises great improvements for *in-memory DBMS*s — as long as the majority of transactions are read-only — it is still susceptible to failures. To improve this aspect fault-tolerance mechanisms must be added. Keeping in mind that we want to increase performance, the fault-tolerance mechanism to be added should also be efficient. An efficient fault-tolerance has two characteristics: i) efficient error detection; and ii) efficient error recovery. The goal of this work is to add fault-tolerance to MacroDB, by implementing state summarization and state recovery for the two in-memory database implementations ran at the replicas: H2 Database, and HSQL Database.

The goal of this work is to add fault-tolerance and healing of software component replicas to MacroDB. We now discuss the key ideas that were explored to achieve this goal.

If we assume a byzantine fault model for software components, to detect faults it is necessary to run operations in more than one node and compare the results before returning the final result to the client. In the context of RepComp, support for detecting these faults have been developed. In this case, it is obvious that the state of a replica has become incorrect. However, a replica may fail silently by getting internally corrupted before returning incorrect results.

This challenge is approached by creating a state summarization for the replica, in the context of this work, we implemented an abstraction of the internal state for the types of replicas used: *H2 Database* and *HSQL Database*.

The main idea here is to use Scalable Counting Bloom Filters to achieve the state summarization. Integrating this structure with the databases we generate a *token* that represents the internal state of the database that can be used by the replicated service to identify divergent replicas by comparing their *tokens*.

Having a way for the internal state of the replica to be acquired is the first step in the process of adding support for fault-tolerance. After a replica has been marked as faulty, the system has to trigger the recovering process for the replica.

This process of recovering a replica is done by repairing the state of the replica, which is done by replacing its state with the one from a known correct replica, hence the need for the existence of mechanisms that allow a replica's state to be exported, and to provide a way for a faulty-replica to import a correct state.

Given that the state summarization *token* for both databases is based on the contents of the that database's tables, updating the state of the recovering replica can be done by recreating those tables in the replica, which is done by recreating the data contained in the tables. Similarly to what was done for exporting and importing the state summarization *token*, exporting the contents of the database is done by running specific functions already offered by the both databases' interfaces.

These functions allow for the contents of the database, more concretely the tables and their respective data, to be exported to a file. The resulting file contains all the database's schema, and all of the tables' data in the form of sql inserts statements. The state of the replica can then be brought up-to-date by parsing and executing the operations contained in the aforementioned file. Both databases provide functions that do this, these functions receive a file with sql statements, and executes them.

Although the replica's tables have now been successfully brought up-to-date, the structure used to summarize the state of the replica — Scalable Counting Bloom Filter — is still out-of-date, and it too needs to be brought up-to-date. This problem is addressed by implementing functions in the replica's that allow for its state to be exported, and for the state of a known correct replica to be imported. After the previous operations has been completed, the next step to successfully recover the replica is to import the

Scalable Counting Bloom Filter from a known correct replica. The replicated service has one of the correct replicas export its filter, and passes it to the recovering replica. Upon receiving this filter, the recovering replica replaces its filter with the imported filter, and the process of recovering the replica ends, as it as successfully been recovered.

1.4 Main Contributions

This works makes the following contributions:

- State summarization for H2;
- State summarization for HSQL;
- State recovery for H2;
- State recovery for HSQL.

1.5 Organization

The remaining document is organized as follows: the next chapter presents the current systems and techniques that in some way or another relate to the topic of this work. Chapter 3 explains how to summarize the state of a service, and Chapter 4 goes a step further and details how can this be applied for replicated services. Then follows a Chapter that goes through the process of State Recovery. And the document ends with an evaluation of the developed work, and the final chapter is dedicated to making final remarks and possible future improvements that can be made to this work.

2

Related Work

2.1 Key Concepts

We start by introducing some base general concepts before analyzing the current related work.

2.1.1 State Machine

As defined by Fred B. Schneider: "A *state machine* consists of *state variables*, which encode its state, and *commands*, which transform its state. Each command is implemented by a deterministic program; execution of the command is atomic with respect to other commands and modifies the state variables and/or produces some output. A client of the state machine makes a request to execute a command. The request names a state machine, names the command to be performed, and contains any information needed by the command. Output from request processing can be to an actuator (e.g., in a process-control system), to some other peripheral device (e.g., a disk or terminal), or clients awaiting responses from prior requests" [6].

2.1.2 State Machine Replication

State machine replication is the replication of a *state machine*. There are a couple of proprieties that the system must guarantee before state machine replication can be applied:

- **Deterministic execution.** If two replicas execute the same sequence of commands in the same order, they must reach the same state and produce the same output [7].

- **Sequential execution.** Before attempting to execute any further operation, the replica needs to finish executing the current operation.

A system can implement state machine replication by executing the same sequence of operations in all replicas. In this work, state machine replication is applied to the realm of multi-core processors, allowing for an improved performance [7, 4] and/or fault-tolerance [8, 9]. However, the solutions proposed in this thesis could be used in any context where state machine replication can be used and even in more general contexts that use replicated state.

2.1.3 Dependability

"Dependability is an integrative concept that encompasses the following attributes: availability: readiness for correct service; reliability: continuity of correct service; safety: absence of catastrophic consequences on the user(s) and the environment; confidentiality: absence of unauthorized disclosure of information; integrity: absence of improper system state alterations; maintainability; ability to undergo repairs and modifications" [10].

In the context of this work, a system is considered to be dependable — provides dependability — if despite any faults, it continues behaving as expected.

2.1.4 Fault-tolerance

Fault-tolerance is one of several ways to attain dependability. Its purpose is to preserve the correct service operation despite any active fault.

The fault-tolerance mechanism is composed by two components:

- **Error detection.** Identifying when an error happens, in order to act upon.
- **Error recovery.** Recovery involves correcting the state of a faulty component. This consists in reverting its state to a previously correct state and bringing its state up-to-date.

Throughout this work we'll consider a replica to be faulty if it manifests one of the following types of failure:

- **Byzantine failures** are arbitrary faults that occur during the execution. These faults encompass both omission failures and commission failures. This type of failure is hard to detect due to the lack of manifestation, the components affected by these failures can keep executing normally while the system can not really mark them as faulty.
- **Fail-stop failures** will halt the replica's execution process instead of performing an erroneous state transformation that will be visible to other replicas [11].

These faults can have different origins — they can be caused by an error in the application's code, deficient design, or be the result of a malicious attack.

2.2 Techniques

This section introduces some basic techniques used in the systems discussed later.

2.2.1 Proactive Recovery

Proactive Recovery [12] provides improved dependability by taking preventative measures against possible faulty replicas. Independently of their state, replicas are periodically recovered, avoiding replicas to go into incoherent states. This is done because there is anecdotal evidence that there is a correlation between the length of time a replica is running and the probability that it will fail [13]. The longer the replica runs, the higher the probability of it failing.

When a replica undergoes the recovering process, it is rebooted and restarted. Usually, the replica is initialized with an out-of-date correct state previously stored. Next, the replica is brought up-to-date, by having its state replaced with the correct state of the system. The correct state of the system can be determined by a majority vote among all replicas. This is done by comparing the states of all replicas. The correct state is then transfered to the new replica, which proceed executing as a system replica. After this last step, the replica's state is considered correct.

2.2.2 Design Diversity

Design diversity [9] is a technique to achieve fault tolerance [14]. The belief here is that having replicas yielding non-coincident errors will ease the job of detecting failures, as this will avoid that replicas coincidently fail in the same operation. Ensuring that when replicas fail they do so with different errors allows the system to identify faulty replicas.

Another advantage of this system is that the aforementioned property not only ensures that errors are detected, but also that most of the implementations will be able to survive that execution.

There are two variants of design diversity: *diverse design diversity*, and *non-diverse design diversity*.

In the former type of design diversity each replica runs a different implementation of a common specification. This approach has the advantage of providing better failure diversity, because each implementation is independently developed and will have tolerance for different errors. Because of this, it is also harder to integrate, as each implementation may have its own way of presenting its internal state.

The latter, non-diverse design diversity, consists in using different versions of the same implementation. This results in having almost identical tolerances for the same errors, thus having weaker failure diversity. The advantage of this variant is that as all implementations have a common code base, they will have nearly identical ways of representing their internal state, which allows for easier integration.

2.2.3 Speculative Execution

Speculative execution is a general technique used to hide latency [15]. Rather than waiting for the results of a slow operation to arrive, the system may opt to work with a speculated result — the result that it is expected to come from the previous operation — thus improving the performance of the system by avoiding the time consumed waiting for results.

The way speculation works is the following: on a replicated service an application instead of waiting for a result of an operation to arrive, speculates what should be the outcome of that operation. When the application creates a speculative result, it saves its current state — by creating a *checkpoint* — and continues the execution process using that result.

When finally the response arrives, if the speculated result does not match it, the application needs to abort all operations for which that result was used, and needs to be recovered. This involves performing a rollback to the last available *checkpoint*, ensuring correctness of state before being brought up-to-date. Then, the application uses the correct result to re-execute all the affected operations.

If the result received matches the speculated one, the application can discard the *checkpoint* created at the beginning of execution, and continue its execution process.

Speculative execution should only be used when the time wasted creating the *checkpoint* is smaller than the time that the system would have to wait for the correct result to arrive, and only when it is possible for the system to estimate the result of an operation. Otherwise speculative execution would be useless for the system, wasting its resources by creating unneeded *checkpoints* or by triggering recovery processes.

A byzantine fault-tolerant system's client can improve its response time by applying speculative execution after receiving the first result, as in the normal case all replicas are correct and the first result equals the result that is obtained in the consensus process.

One thing that needs to be avoided is the externalization of speculative results, ensuring that the state associated with a speculative result is never committed, because by externalizing the result of a speculative execution, the system would not be able to revert the effect of that result.

An associated concept is the *boundary of speculation*. This boundary is what allows the system to control the repercussions of a bad speculation, by limiting the number of operations that are affected by a miscalculated result.

The larger the boundary allowed by the system, the higher the risk of having to take recovery measures. This means that having large boundaries have higher costs for the system in the case of miscalculated results, as more operations are tainted by that result. The advantage of having large boundaries is that they allow for improved throughput, because more operations are allowed to use speculative results. On the other hand, if the system uses very small boundaries, speculative execution will not yield a great improvement in throughput because there will be less speculative operations allowed to take place. This also implies that the number of *checkpoints* that will be created is greater and the number of state verifications that take place will also be greater, thus restricting the throughput of the system.

2.3 Systems

This section presents related work. Although our focus is on systems that rely on replication in systems with multiple cores, we also present other relevant works. We start by presenting systems that try to improve performance and later those that focus on fault-tolerance.

2.3.1 Systems for improved performance

The following systems were developed for improving performance. The first, Multimed [16], provides improved performance by treating multi-core processors as a distributed system. The second, Eve [17], provides support for multi-threading in state machine replication systems. Lastly, Macro-Component [5, 18], exploits the performance differences of various components, providing the best performance for all operations.

2.3.1.1 Multimed

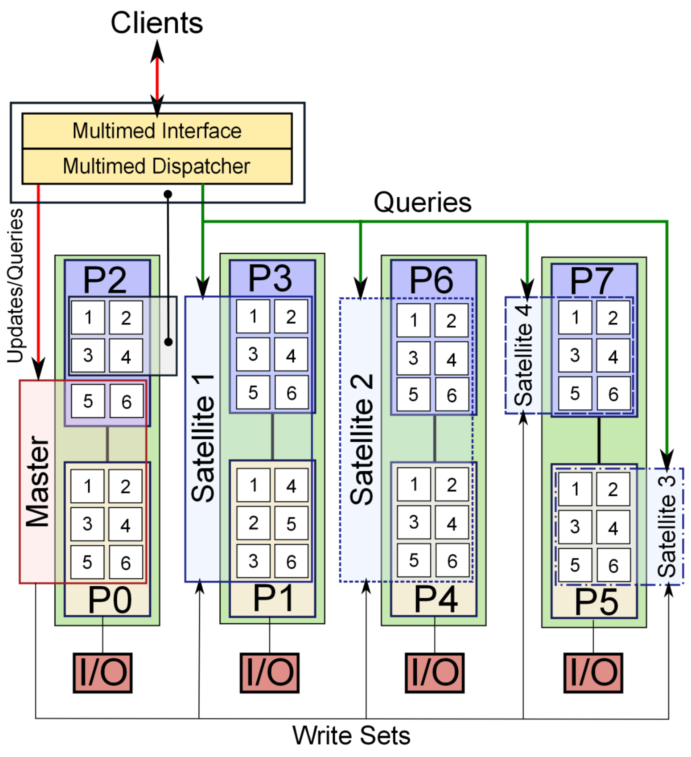


Figure 2.1: A possible deployment of Multimed (Figure taken from [16])

Multimed [16] is a database system, based on Ganymed [19], for multi-core computers.

Database management systems seem to not be able to take full advantage of multi-core processors, thus being unable to improve their performance solely by running on these processors. Multimed addresses this problem by exploiting the master/slave architecture (Figure 2.1). It divides the multi-core processor into several partitions — each containing at least one core — and then assigns one replica to each partition. One of the replicas acts as the master while the others are treated as slaves (*satellites*).

Another important aspect of this architecture is that satellites can either keep a full copy of the master's database or a partial copy. Having a full copy of the database has the advantage of reliving Multimed from having to keep records for the data kept at each of the satellites, but also incurs in higher synchronization costs, as updates are more frequent and the volume of data that they contain is larger.

To ensure synchronization between replicas, all write operations are only done at the master replica. This replica then propagates the changes to all of the satellites. This means that satellites are not always kept up-to-date, as modifications committed at the master have first to arrive and then be committed.

When the master commits changes to the database, these are propagated to all satellites — or to a fraction of satellites, in the case of partial replication — as a collection of rows. The satellites enqueue these updates, and apply them in the same order they were committed at the master. Multimed enforces snapshot isolation [20].

Contrary to what happens with write operations, this allows Multimed to redirect read operations to any of the satellites, where read-only transactions can execute without coordination with other satellites. This allows the load to be distributed among all database replicas. The dispatcher has the responsibility of routing operations. When a read-only query arrives it is routed to the most lightly loaded satellite capable of running the query, thus improving the throughput of the *DBMS* by balancing the workload. If no suitable candidate is found, the *dispatcher* can take one of two actions: i) wait for a suitable candidate to appear; or ii) elect the master replica to run the query.

This works even if the satellite is not synchronized with the master's state, because it has a queue of the modifications pending to be applied, and only has to commit those that guarantee that its state corresponds to the master's state at the moment the operation was received.

The distributed approach taken by Multimed is based on the intuition that its better to have various *DBMS* engines using all the available resources than to have a single engine using all resources. Given its distributed nature, Multimed allows *DBMS*s to exploit multi-core processors, giving these systems improved performance. This system is similar to MacroDB [5], as both cases exploit the performance aspect that replication by having each of their replicas running a *DBMS*.

However, this system does not incorporate any mechanism to recover failed satellites. Multimed only guarantees that no data is lost due to a satellite failure, as all data is durably committed by the master.

2.3.1.2 Eve

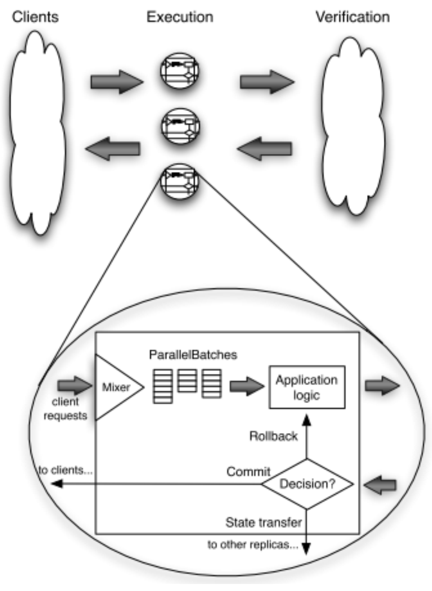


Figure 2.2: *Eve*'s architecture (Figure taken from [17])

The main goal of Eve [17] is to improve the performance of systems running state machine replication protocols on multi-core machines. The main idea is to change the way traditional state machine replication operates. Instead of agreeing on the order of execution, the system agrees on the output of execution. Having only to agree on the correctness of the results, the system can exploit the multi-threading capabilities of the multi-core processors. This allows the system to scale with the available number of cores, resulting in an overall improvement in performance. The architecture of this system is illustrated in Figure 2.2.

There are three proprieties that need to exist in order for Eve to accomplish its goal:

- **Divergence is uncommon:** the most common scenario at the end of executing an operation should be that replicas have converged;
- **Efficient state comparison:** there needs to exist an efficient mechanism to identify the state of a replica;
- **Efficient replica repair:** repairing a faulty replica should be done in the most efficient way possible.

Guaranteeing that convergence is common is done by the *mixer*. This component takes the requests that arrive at the system and groups them into batches of non-conflicting requests, which allows replicas to execute them in parallel. While batches are executed sequentially as in traditional state machine replication. The *mixer* knows which requests will conflict by reading its configuration file. This file can be something as simple as a list of transactions used in the application, stating the tables they access and the type of operation that it is performed (read/write). Each replica can exploit multi-threading for the execution of a *batch-parallel* request.

After executing a *batch-parallel* request, the results and the state of the replica are compared with the ones obtained by the other replicas. Instead of comparing the result and state itself — as this would entail sending the whole result and state — only a token representing the replica's state is sent. This token is the root-leaf of the *merkle tree* used by the replica to store its objects. Using a *merkle tree* to generate a representation of the replica's state is challenging. For this to work the system must devise a way to ensure that each replica constructs its tree deterministically.

Eve overcomes this challenge by having each replica construct its tree only at the end of execution, before generating the token. The intuition here is that replicas that executed correctly should have the same objects to add to the tree, and by postponing the creation of the tree until the very last moment, they will construct the same tree.

If the state of the replicas is different — i.e., the tokens did not match — this means that some of the replicas have diverged. To identify which replicas diverged the system runs a consensus algorithm with the collected tokens. If there are enough equal tokens, that means that this token corresponds to the correct result. All replicas whose state yield a different token are considered to have diverged, and need to be repaired.

Repairing a faulty replica involves undoing the operation that caused divergence, and bringing the replica to the correct state. A rollback is performed to undo the results of the operation, leaving the replica in a correct but out-of-date state. The rollback process is done by having the pointers of the *merkle tree* point to the original object. This ensures that the replica is in a previously correct state. The last step involves bringing the replica up-to-date. This is done by sequentially re-executing the batch that caused divergence, in order to guarantee progress.

Some of the techniques used in Eve were applied in this thesis, namely the techniques related to state summarization. The state summarization process implemented in our prototype makes use of the efficient way that the *merkle tree* has for performing state summarization. Although the *merkle tree* is not used in the same manner that it is used in Eve to achieve state summarization of the replica, it still proved to be advantageous to integrate this structure in the implemented state summarization process, as it provides a more efficient way for state summarization, in our case, for summarizing the state of the structure used to achieve this, as described in subsequent chapters.

2.3.1.3 Macro-Component

The goal of Macro-Component [4] is to provide better performance or fault-tolerance over standard components, doing so by applying diverse component replication. With Macro-Component, software components in a program are replaced by a Macro-Component that includes several replicas of the same specification, typically with different implementations.

The intuition behind Macro-Component is that no single implementation of a software specification yields the best results for every operation — i.e., a component that has the fastest results for some operations may have the slower results for other operations. This creates a dilemma for the programmer, forcing him to choose the component that is the best fit for his application, thus sacrificing performance for some of the operations. This decision must be made by taking into account the expected workload for the application. The problem is that it is not always possible to know what will be the workload, as many applications vary their workload type throughout execution, and making the correct choice in these types of scenarios is impossible. Macro-Component removes this decision from the design processes, allowing the programmer to use the fastest method for any given operation.

For providing better fault-tolerance, Macro-Component can explore different component implementations to mask component bugs.

When a method is called on a Macro-Component, one of two paths is taken: i) if the method is expected to return results, a synchronous invocation model is used, concurrently calling the same method on all replicas — when configured to improve performance, the first result obtained is returned to the application; when configured to improve fault-tolerance, the result is returned after being confirmed by more than one replica; ii) in the case of the method not yielding any results, tt is known that the result will be ignored by the application, an asynchronous invocation model is used, concurrently executing this method alongside the application's thread, further improving the throughput over standard components.

One limiting factor is the need for replica consistency. Although necessary, it negatively affects the system's performance, as synchronization will reduce the amount of parallelism that can take place. Macro-Component uses a variant of the master-slave scheme, that uses a master-rotating process. After executing a write operation, the current master is demoted to slave, and the first slave to complete all pending synchronization operations is elected as the new master. During this time, read operations are routed to the former-master. This forces a total execution order for write operations in all replicas, thus guaranteeing that they converge.

Macro-Component shows that replacing standard components with their Macro-Component siblings, yields improvements in performance, as this approach exploits the performance differences offered by the various implementations. The system includes no mechanism for recovering replicas, which is the focus of this Msc work.

2.3.2 Systems for improved fault tolerance

The following systems were developed with dependability as their main goal. The first, N-Version programming [9], achieves this by using various implementations of the same software specification, which will guarantee failure diversity. The second, BASE [8], uses abstraction to provide an improve byzantine fault tolerant system. The third, Frost [21], forces replicas to have complementary schedules in order for the application to be able to evade data-race bugs. Lastly, Respec [22], uses speculative execution and external determinism, to provide online deterministic replay for multi-core processors, which allows the system to improve its dependability.

2.3.2.1 N-Version Programming

N-Version programming [9] (NVP) is a technique that aims to improve a system's fault tolerance mechanisms, thus improving dependability.

The main idea behind NVP is to force replicas to have non-coincident failures (failure diversity), allowing the system to detect faulty replicas by comparing their outputs of execution.

This is achieved by running N implementations of a given software specification. Running various implementations of the same specification guarantees that any errors that occur will be as diverse as possible. Having replicas yielding non-coincident errors allows the system to detect faulty replicas, that are then to be recovered.

The system works as follows: the results of each execution are sent to a decision algorithm which, by consensus, derives the correct output of execution. The criteria used to elect the correct result can be adjusted, favoring throughput or dependability. The larger the number of replicas that have to agree on the output of execution the slower the decision process, but gives the system better dependability. The lower the number, the higher the throughput of the system, at the cost of risking using incorrect results in future executions.

Afterwards, the correct result can be used to derive which replicas went inconsistent during execution. Replicas that yield incorrect results, are considered to be faulty, and need to be repaired.

In order to successfully integrate this technique in a system, we need to guarantee a couple of things. Firstly, a clear software specification that states the behavior to be expected throughout execution. This specification should leave no room for doubt. Secondly, the system must have some mechanism to detect inconsistent replicas. And lastly, a mechanism capable of repairing faulty replicas. One example of a system that applies this technique is the Macro-Component [4], where different replicas are allowed to use different implementations of the same software specification. This allows for an improvement in performance, as different implementations will outperform other implementations for some operations.

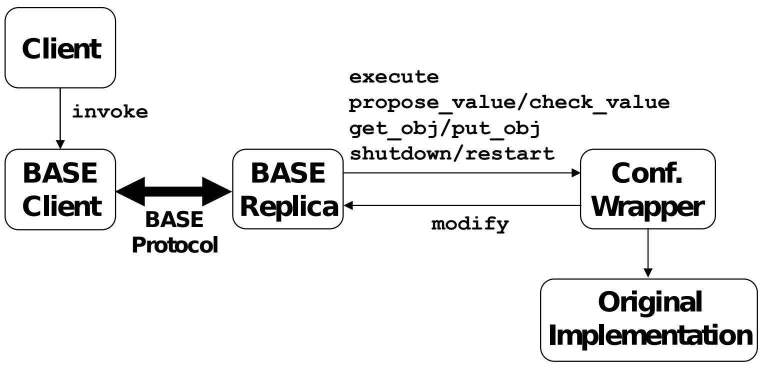


Figure 2.3: BASE functions overview (Figure taken from [8])

2.3.2.2 BASE

BASE [8] is a distributed replication system designed with the objective of providing improved fault-tolerance, focusing mainly in byzantine faults. This system uses opportunistic N-version programming to achieve those goals.

This system combines abstraction with a byzantine fault tolerance (BFT) library to improve the system's ability to mask software errors. The original PBFT library [23] has limited use, as it requires that all replicas run the same implementation of a software specification and that they update their state deterministically. Therefore, it can not tolerate errors that cause all replicas to fail concurrently.

BASE solves this problems by implementing mechanisms that allow replicas to run different implementations or non-deterministic implementations. The main concepts behind this system are the *abstract specification*, the *abstract function* and the *conformance wrapper*. The *abstract specification* specifies the *abstract state* and the behavior of each operation, describing how these operations affect the state of the replica. The *conformance wrapper* is the component that ensures that each implementation behaves accordingly to the *abstract specification*. Finally, the *abstract function* is the component that allows to map from the concrete state of each implementation to the *abstract state*, and vice-versa by implementing one of the inverse functions of the *abstract function*, thus allowing for a replica to be recovered. Figure 2.3 shows an overview of BASE's functions.

With these components BASE is capable of reusing existing code. Different off-the-shelf implementations of a service can be used, as the *conformance wrapper* ensures that each implementation behaves accordingly with the *abstract specification*, despite the fact that most implementations fail to conform with their own software specification (e.g., many database management services exist, but they behave differently from each other).

This system also uses proactive recovery [12], rebooting replicas, restarting then from a clean state, and then bringing them up-to-date using an *abstract state* obtained by consensus from the other replicas. The aforementioned *abstract state* is created by identified the core objects that represent the internal state of a replica. This means that independently of the replica's implementation, there are concepts — in the form of objects — that are transversal to all implementations, therefor providing the needed basis for creating an *abstract state* that can be used by any implementation.

In this thesis, the idea of having an *abstract state* that could be used for summarizing the internal state of a replica — independently of its implementation — was used. This allowed for different replicas to run different database implementations, while still providing a way for comparing the internal state of two different implementations in order to determine possible state divergences that might have occurred during operation. Additionally, the technique used to recover replicas used by BASE — proactive recovery — is also used in the implemented prototype. Recovering a replica that was deemed to be faulty is done by rebooting the replica, leaving the replica in a correct but out-of-date state, and then using the correct state of any of the other replicas to bring the recovering replica up-to-date.

2.3.2.3 Frost

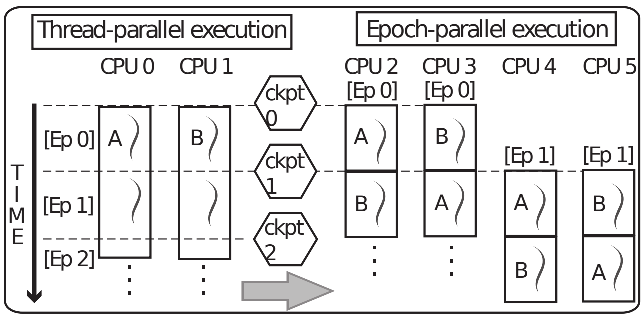


Figure 2.4: Frost Overview (Figure taken from [21])

Table 2.1: Frost Plan of Action (Adapted from [21])

Epoch Results	Survival Strategy
A-AA	Commit A
F-FF	Rollback
A-AB/A-BA	Rollback
A-AF/A-FA	Commit A
F-FA/F-AF	Commit A
A-BB	Commit B
A-BC	Commit B or C
F-AA	Commit A
F-AB I	Commit A or B
A-BF/A-FB	Rollback
A-FF	Rollback

The left column shows the possible combinations of results for three replicas; the first letter denotes the result of the thread-parallel run, and the other two letters denote the results of the epoch-parallel replicas. F denotes a self-evident failure; A, B or C denote the result of a replica with no self-evident failure. The same letter is used when replicas converge.

Data races are a common source for bugs in multi-threaded applications, creating the need for mechanisms that are able to deal with such bugs. Frost [21] is a system that aims at finding concurrency bugs in multi-threaded applications. The idea behind Frost is to force the execution of complementary schedules.

Using complementary schedules tries to ensures that at least one of the replicas avoids the bug, by running the instructions that lead to that bug in a different order.

Frost achieves this by running each thread on its own processor, and with *epoch-parallel* execution — dividing the program execution in time-slices (*epochs*), and having each replica (*epoch-parallel replica*) run these *epochs* concurrently (see Figure 2.4).

This initial approach requires threads to run in a *non-preemptively* manner, making this solution only suitable for single-core processors, as this architecture allows Frost to control the schedule used by the replicas.

In order to apply the same ideas to the realm of multi-core processors, Frost uses a third replica. The third replica runs its threads in parallel (*thread-parallel replica*), which allows Frost to identify points during execution where system calls and synchronization events take place, and speculatively create *checkpoints*. The *checkpoints* are then used to divide the execution in synchronization-free regions (*epochs*), thus guaranteeing that only a data-race can take place during an *epoch*.

The fault-tolerance mechanism works in two phases. In the first phase, Frost starts by checking if the replicas experienced *self-evident* failures. If they do not show signs of *self-evident* failures, it looks for signs of failure by comparing their state at the end of the *epoch*, and the results yielded during that *epoch*.

After identifying that a race took place (i.e., a replica failed) and in order to survive execution, Frost chooses the survival strategy that is most likely to have at least one replica survive.

This is done by analyzing the results of the *epochs*, and depending on the results yield by replicas it chooses the plan of action. As shown in Table 2.1, Frost may choose to perform a rollback, or to commit the results from a correct replica.

It decides to commit a result if all replicas converged, or if the combination of results allows for a correct result to be identified. If a *epoch-parallel* replica is chosen to be committed, all subsequent *epochs* generated by the *thread-parallel* replica are invalid, and need to be discarded. Then, the system starts new *thread-parallel* and *epoch-parallel* replicas using the committed state.

The rollback process occurs when no correct result can be identified. In this case, Frost may choose to generate additional replicas to further study the *epoch* that led to a divergence. The additional replicas are started from the *checkpoint* created at the begging of the *epoch*, and their results will allow Frost to choose which replica to commit.

In order for replicas to continue their execution process, they must defer the output of the previous *epoch* (speculative execution) — as *Frost* will not externalize any output without it being committed. There is a trade-off between performance and dependability, as longer *epochs* provide better dependability, shorter *epochs* provide better performance.

Frost's approach to data-race bug detection and survival can be applied to MacroDB [5]. This would allow for the *DBMS*s used by MacroDB to have more relaxed locking mechanisms, while maintaining correctness of execution. By integrating the ideas behind Frost in the development of MacroDB's fault-tolerance mechanism, the system would be capable of detecting and surviving data-race bugs, thus further improving its resilience to failures.

2.3.2.4 Respec

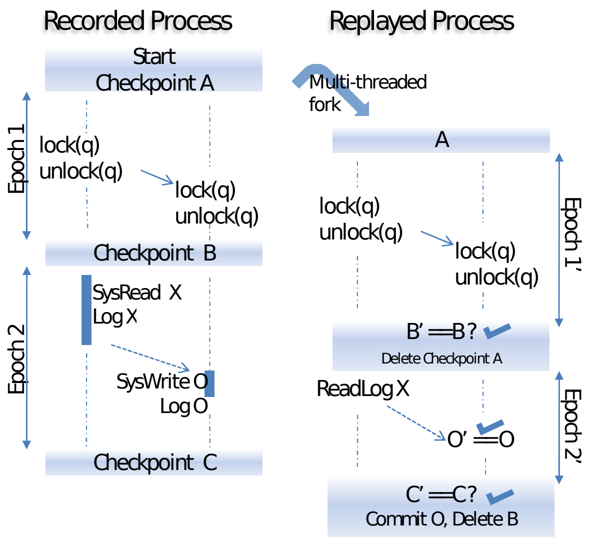


Figure 2.5: Example of a Respec execution (Figure taken from [22])

Respec [22] aims to improve the performance of online replay on multi-core processors. Deterministic replay is used to record and reproduce the execution of a program. In the online variant of deterministic replay, both recorded and replayed processes run concurrently. This allows the system to improve its dependability, as the recorded process can be used as a backup when the replayed process fails. The idea behind Respec is to reduce the overhead added to the system by using speculative logging (it only logs information that is needed to ensure deterministic replay), and externally deterministic replay — which relives the system from having to constantly ensure that the processes' state match during execution, only having to converge at the end of execution.

Figure 2.5 depicts Respec's execution. At the beginning of program execution, Respec checkpoints the *Recorded Process*, and forks the *Replayed Process*. Then at predetermined intervals Respec checkpoints the *Recorded Process*, effectively dividing the program execution in various regions (*epochs*). The insight here is that if during execution the states of the processes do not match, they are likely to diverge at the end of the execution process. Dividing the program execution in various regions allows Respec to perform periodical convergence checks, and take early recovering measures if the processes do not match. Speculative execution is also used in the *Recorded Process*, thus improving its throughput, as it does not have to lock waiting for results in order to continue execution.

During an *epoch*, the *Recorded Process* logs its system calls. When the *Replayed Process* reaches a point during execution where a system call is made, instead of executing it, it emulates the system call using the logs created by the *Recorded Process*. This works because the *Recorded Process* executes ahead of the *Replayed Process*, as it does not have to perform checks during each *epoch*.

Respec detects divergences — mis-speculations of race-free regions — in two ways. The first involves comparing the arguments passed to the system calls and the resulting output, allowing for an early recovery of the *Replayed Process*. The second method used by Respec to detect divergences is to analyze the state of the processes' at the end of the *epoch*, triggering the recovery process if they do not match.

The recovering process involves rolling back both processes to the beginning of the *epoch*, leaving them in the same correct state. Then, Respec optimistically re-executes the problematic region, and performs the same checks as before. On repeated failure, in order to guarantee execution progress, Respec switches to the uniprocessor execution model — executing one thread at a time until both processes reach a system call. After this, the processes' state match and parallel execution can resume.

State machine replication systems can exploit the concepts behind Respec to improve their resilience to software errors. The replayed process can be used as a primary replica, and the recorded process as a secondary replica. Then the error detection process used in Respec, can be used as an efficient way to keep the state of both replicas synchronized. In the advent of a primary failure, the secondary replica can takeover the execution while the system recovers the faulty replica.

2.4 Examples of replicated services

This section discusses the challenges posed to integrate n-version programming with state machine replication in database systems. Although the used techniques are a refinement of the techniques presented earlier, this work is particularly relevant to our work as we will address the problem of healing replicas in MacroDB [5], a macro-component system that replicates in-memory database systems.

2.4.1 Off-The-Shelf SQL Database Servers

The poor dependability shown in off-the-shelf (OTS) SQL database servers [24, 25] is due to the fact that they assume that only two kinds of errors can occur: *fail-stop*, and *self-evident*. *Fail-stop* errors are those that cause the software to crash, and *self-evident* errors are those that are externalized, for example, in the form of error messages. This assumption is not correct, as there are other kinds of errors that can affect the application's behavior, and when they manifest themselves, the system will misbehave, as it is not prepared to deal with them.

One possible solution for improving dependability in OTS SQL database servers is to use active replication. Active replication consists in comparing the results throughout execution, in order to determine if some error took place. Comparing the results of each execution will determine the plan of action, if one replica shows a result that is inconsistent with the results obtained by the other replicas, we can assume that the replica has failed, and needs to be recovered. Recovering a faulty replica can be done by copying the state of a correct replica over the state of the faulty replica. But this approach will only work if the majority of replicas executed correctly, allowing the consensus decision algorithm to infer what is the correct result. In the case of a majority of replicas failing with the same incorrect result — thus misleading the decision algorithm into taking for correct an incorrect result — all the other replicas (including the replicas that had the correct result) will be forced to converge to the state of a faulty replica. From this point on, the system state is incorrect, and all operations will yield incorrect results.

This means that the former solution is not enough to guarantee better dependability. The system needs to ensure that when replicas fail, they fail with errors as diverse as possible. Failure diversity, more specifically design diversity, will ensure this property [9], avoiding replicas to have coincident failures. There are two types of design diversity: non-diverse, and diverse. The former is achieved by using evolving versions of the same database product, has the advantage of being easier to integrate, but will not yield any significant advantage to the detection non-self-evident/non-crash failures, as versions of the same product will almost likely have coincident errors. The latter, consists in using different database products, and improves greatly the detection of non-self-evident/non-crash failures, but it is harder to integrate.

Applying diverse design diversity is more challenging for three main reasons:

- product-specific SQL dialects: each database may have its own implementation;
- missing/proprietary features: each database product may offer features that are not present in the others;
- replica consistency among all implementations.

It is possible to overcome the first challenge by doing on-the-fly translations of SQL statements. The second can be partly solved in two distinct ways: i) using only the common subset of features; or ii) trying to emulate some of the missing features (e.g., by rephrasing SQL statements). The third problem — replica consistency — can be solved by applying the divide-and-conquer methodology, dividing the problem in two smaller ones: detection, and recovery.

To identify a faulty replica the system can do one of two things: i) compare the results of read operations, and verify if they match; or ii) verify the list of affected records after write operations, and check if there are any differences between the replicas. After comparing the state of the replicas, if the results diverged the system can then use its fault-tolerance mechanism to mask the fault.

The second problem — recovery — is done in two steps. The first step is to undo the operation that left the replica inconsistent, rolling back the transaction to the last available checkpoint. The second step is to bring the replica to a consistent state, this can be done by either re-executing the transaction — hoping that the new result is correct — or by copying over the database contents of one of the correct replicas, thus ensuring that the replica's state is now consistent.

In this thesis similar problems had to be solved. The first, identifying replicas that have become inconsistent during operation and need to undergo the recovering procedure follows a similar logic to the aforementioned one. Although, it is not verified if the list of affected records is the same for all replicas as this would entail sending result sets for each operation, the state of the replicas is compared using a state summarization structure at each replica, and comparing the *token* that is generated by said structure, allowing for divergences of state to be detected. Additionally, the solution for recovering an inconsistent replica implemented in this thesis follows the same line of thought described earlier. Where a replica that is deemed to have become inconsistent is recovered in two steps. The first involving restarting the replica, leaving the replica in a correct but out-of-date state, and then bringing the state up-to-date by transfering the state from a correct replica.

3

Summarizing State Service

This chapter explains how to achieve the summarization of state for a service. The first section presents the Scalable Counting Bloom Filter and all the process that took place to create this structure. The final section explains how to perform state summarization for the Scalable Bloom Filter itself, which is crucial for the use of this structure to be able to efficiently summarize the state of a service.

3.1 Scalable Counting Bloom Filter

The goal for this section is to detail how the Scalable Counting Bloom Filter can be used as a state summarization tool for a service.

3.1.1 Bloom Filter

The Bloom filter is a probabilistic data structure used to store a set of elements in a efficient way. The intuition behind this structure is that it is only needed to store a representation of elements that are added, in order to be able to execute membership operations. This representation can be smaller than the size of the element, making this structure efficient.

The *modus operandi* behind this structure is to use a array of bits to store information about elements added to the set. When a new element is inserted in the Bloom filter, k positions of the array are set to 1, effectively storing a representation of the element. To verify if an element belongs to the Bloom filter, all of the k positions — that correspond to the representation of the element — of the array are checked in order to determine if they are all set to 1. If this is true, the element is said to belong to the filter, otherwise, it not present in the filter.

Depending on the desired false-probability rate set in the filter, the number of positions used in a the array to store the representation of an element varies. The larger the array and the lower the false-probability rate, more positions will be used to represent an element.

Although the false-probability rate can be controlled, there is another problem associated with the Bloom Filter's approach to store elements. As elements are added to the filter, the positions in the array of bits start to be all set to 1, which will have an impact in the false-probability rate. In the extreme case where all of the bits get set to 1, checking for membership of any element will result in the filter claiming that the element is belongs to the set, when it does not. As we will later see, this is a problem related with the fact that the Bloom Filter can not expand beyond its initial size.

$$\rho = (1 - e^{-k(n+0.5)/(m-1)})^k \tag{3.1}$$

The probability of a false-positive — when the Bloom Filter falsely claims to have an element, when in fact it has not — is given by the formula shown in Equation 3.1. This equation gives the false-probability ρ for a Bloom Filter of length m, that has n elements, and that uses k hash functions. This formula is only applicable for scenarios in which it is assumed that the aforementioned variables do not tend to infinity. In such case a more complex formula to calculate the false-positive probability ρ has to be used, which will not be discussed.

The Bloom Filter offers the following operations: i) inserting elements; and ii) searching for elements (checking if an element belongs to the set).

Insert Operation

```
Data: $x$ is the object key to insert into the Bloom filter
Function: insert($x$)
    for $j$ : 1...$k$ do
        $i \leftarrow h_j(x)$;
        if $B_i == 0$ then
            /* Bloom filter had zero bit at position $i$ */
            $Bi \leftarrow 1$;
        end
    end
```

Algorithm 1: Bloom Filter insert function

When inserting an element in the Bloom Filter, only a representation of that element is stored in the filter's array of bits. As shown in Algorithm 1, when adding element x to the Bloom Filter, k hashes are generated. This is achieved by passing element x through k hash functions h_j (with $j \in \{1, ..., k\}$), which generate i positions that correspond to the element representation.

Then the representation of element x is stored in the Bloom Filter by accessing its array of bits at each of the i positions and changing the corresponding values to one if they were set to zero. This effectively stores element x's representation in the array, thus successfully completing the insert operation for that element.

Search Operation

The search operation is similar to the insert operation, as shown by Algorithm 2. When checking if element x belongs to the set, k hashes are generating by passing element x through k hash functions h_j (where $j \in 1...k$). The resulting k hashes are then translated to the i positions that represent the element in the array of bits. The values of these i positions are then checked to see if they have been set to 1. If all positions are to 1, element x is considered to be present in the Bloom Filter with a false-probability rate of ρ. A false-positive occurs when the positions that correspond to element x's representation have all been set to 1 by the other members of the set, thus wrongly indicating to the filter that x is a member.

Figure 3.1 exemplifies a case of a false-positive. Here we see a Bloom Filter that has elements x,y, and z. But when checking if w also belongs to the filter, the Bloom Filter will falsely claim to contain w due to the fact that the positions that result from the translation of running w through the k hash functions overlap the presentations of the elements present in the filter.

Thus, despite w not being present, given that all of the positions that correspond to w's representation are all set to 1, the filter would wrongly claim that w was a member of the set.

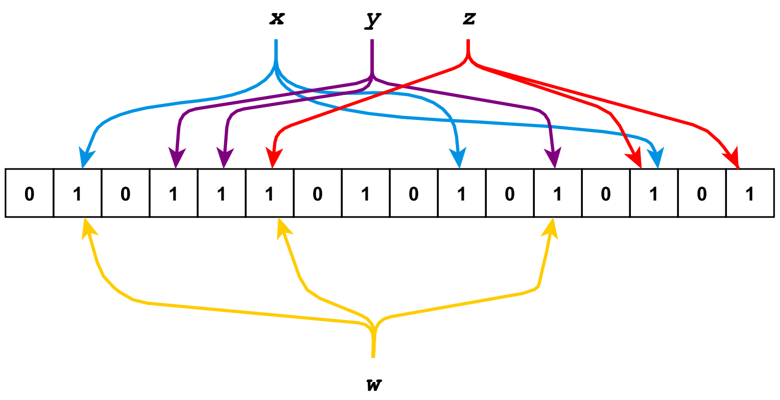


Figure 3.1: Example of elements sharing positions in a Bloom Filter.

The more positions used to represent an element, the lower the probability of having false-positives. Although this is an advantage it comes at a cost, as there are more positions in the filter used to represent an element, the less elements can be stored in the filter.

```
Data: $x$ is the object key for which membership is tested.
Function: search($x$)
    $m \leftarrow true$;
    $j \leftarrow 1$;
    while $m == true$ and $j \leq k$ do
        $i \leftarrow h_j(x)$;
        if $B_i == 0$ then
            $m \leftarrow false$;
        end
        $j \leftarrow j+1$;
    end
    return $m$;
```

Algorithm 2: Bloom Filter search function

The fact that positions of the array of bits are shared amongst elements still have their values set to one, which does not convey any information about how many elements are sharing a given position, restricts the types of operations supported, as removing an element from the Bloom Filter is not possible. If an element was to be removed, given that it might have positions shared with other elements, it would negatively impact the filter's ability to operate correctly, as the removal of that element would lead to all of the positions that were used to represent the element in the array be set to zero. This means that the remaining elements that share some of those positions would be also removed from the Bloom Filter, as their representation in the array would also be affected, thus the filter would wrongly claim that those elements are not present, when they are.

Another draw-back of the Bloom Filter is its un-scalable nature. The fact that the length of the filter has to be set prior to its use — during the initialization process — associated with the relation that exists between the hash function h_j and the length of the filter, restricts its growth. This is due to the fact that the structure only stores a representation of the elements and not the elements themselves, restricting the filter's length from being modified as the elements that were used to populate the array of bits are no longer present, thus they can not be used to populate a filter of bigger size.

Figure 3.2 depicts the possible representation of the same element x in filters of different sizes. Both filters have element x's representation stored. In the filter with length 8 — left side — this representation is achieved by the bits of the array at positions 1,4 and 7. In the filter of length 10 — right side — the element's representation is performed by setting the bits at positions 2, 6 and 8. As we can see filter's with different lengths generate different representations for the same elements. Despite both filters using 3 bits to store x's representation, we can see that the bits used by both filters vary, which justifies the inability of the Bloom Filter expand beyond its initial size, as there is no information stored about the elements themselves, but only a representation.

Despite the filter having different lengths, the positions used to represent x also differ, this means that if we consider that a filter of length 8 was expanded to a length of 10, and

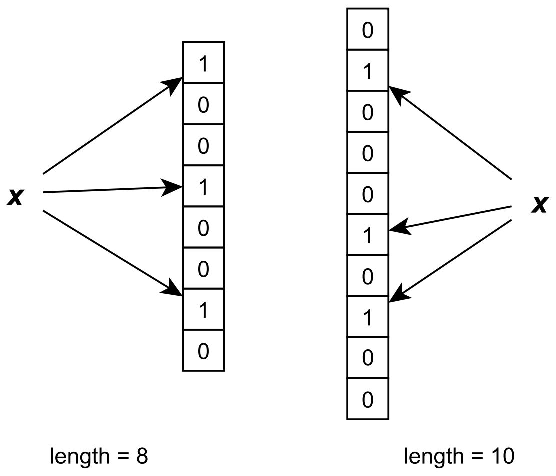


Figure 3.2: Comparison between two Bloom Filters of different lengths

that element x was already present in the filter. If x was tested for membership after the expansion of the filter, the filter would claim that the element was not present, as all of the hashes generated by the hash function h_j would be translated to different positions, thus when checking the values of those positions, given that they are not all set to 1, the filter would have no way of knowing that x was present.

This problem can not be solved by converting the old representations to the new filter's length, as the filter has no knowledge about the elements that were added. The filter's only function is to store elements' representations, which negates the idea of converting old element representations into new ones.

3.1.2 Counting Bloom Filters

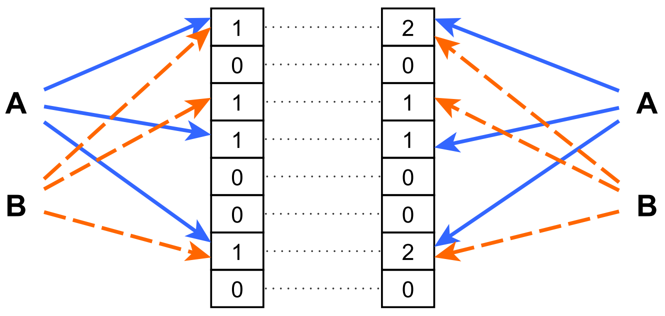


Figure 3.3: Comparision between Bloom Filter and Counting Bloom Filter

Counting Bloom Filters were proposed as a solution for one of the inherent problems of the Bloom Filter: its inability to support element removal [26].

The idea behind this modified version of the Bloom Filter is to have the structure used to store the representation of the elements to be capable of supporting values other than zero and one. This enables the Counting Bloom Filter to keep track of how many elements are sharing any given position, which is the basis for adding support for element removal. The intuition behind this is that by providing the means for each position to vary between 0 and n (where n is the biggest value that can be represented by the position), removing an element can be achieved simply by decrementing the values that correspond to the element to be removed. This approach has no impact in the remaining elements that have some positions shared with the removed element because their representation in the structure still remains.

Figure 3.3 exemplifies the core difference between these two structures. On the left side we see how a Bloom Filter would store elements A and B, and on the right side we see how a Counting Bloom Filter would store the same exact elements. The difference that can be seen is that positions that are shared between A and B are represented in different ways in both structures. While the Bloom Filter does not make any distinction between a position that is shared between elements from one that is not, the values are either 0 or 1. However, for this same scenario the Counting Bloom Filter distinguishes positions that are shared from those that are not. Those positions that are shared between A and B are set to 2, while the other positions that are storing these element's representations are set to 1.

Typically this is achieved by replacing the *BitSet* that is used in the Bloom Filter with another structure — byte array — or by using blocks of bits in the *BitSet* as a single position, which allows for each position to vary between 0 and n. When using the former approach, each position is only limited by the numbers that can be represented with a single byte. Since a position has two possible states, them being the position is being used to represent elements or it is not. This means that the minimum value that we will need to support is 0. This means that the byte array will be supporting un-signed integers, which implies that the maximum value for each position is also the maximum integer value that can be represented with a single byte: 255. By having each of the positions in the array vary between 0 and 255, this means that at most 255 elements can be sharing a single position.

Although the aforementioned approach is able to successfully reach the goal of enabling the Counting Bloom Filter to support element removal, supporting 255 values for each position seems excessive. The scenarios that would have a single position being shared by such a large number of element's representations are so small that supporting that wide range of values would be wasteful. It has been hypothesized by some authors that using 4 bits — half a byte — to represent a single position was enough [27], as this allows for each position to be shared by 15 different elements. Additionally, if collisions start to be too frequent this means that the filter is becoming full, and by allowing further elements to be inserted the filter will no longer respect the false-positive probability.

Considering that the Counting Bloom Filter is based on a Bloom Filter, it offers the same operations supported by that structure — element insertion, and element lookup — and adds support for element removal.

Insert Operation

Algorithm 3 depicts the logic behind this operation. Comparable to the insert operation of the Bloom Filter, when inserting an element x in the Counting Bloom Filter k hashes are generated. This is done by running element x through a hash function h_j (where $j \in 1...k$). The k hashes resulting from that operation are then translated to the positions in the byte array that will be used to store element x's representation.

For each i position in the byte array, its new value B_i' is set to be the old value B_i plus one, which means that each position that corresponds to element x's representation is increased by one.

```
Data: $x$ is the object key to insert into the Counting Bloom filter
Function: insert($x$)
    for $j$ : 1...$k$ do
        $i \leftarrow h_j(x)$;
        /* Verify that incrementing the position's value does
           not result in an overflow                          */
        if $B_i < 14$ then
            $B_i' \leftarrow B_i + 1$;
        end
    end
```

Algorithm 3: Counting Bloom Filter insert function

Search Operation

The search operation in the Counting Bloom Filter is identical to the one of the Bloom Filter, as depicted in Algorithm 4.

```
Data: $x$ is the object key for which membership is tested.
Function: search($x$)
    $m \leftarrow true$;
    $j \leftarrow 1$;
    while $m$ == true and $j \leq k$ do
        $i \leftarrow h_j(x)$;
        if $B_i == 0$ then
            $m \leftarrow false$;
        end
        $j \leftarrow j$+1;
    end
    return $m$;
```

Algorithm 4: Counting Bloom Filter search function

When checking the Counting Bloom Filter for membership of element x, k hashes are generated by running element x through the hash function h_j (where $j \in 1...k$). Then, these k hashes are translated to the positions of the byte array that represent element x.

The Counting Bloom Filter accesses each i position, and verifies if the value B_i differs from 0.

If all B_i values are verified to be greater than 0, element x is considered to a member of the set (with a false-probability rate of ρ), thus it is present in the Counting Bloom Filter and the operation returns $true$. However, if any of the values B_i is 0, this means that element x is not contained in the set, therefore not be present in the filter, which immediately ends the search operation and the function returns $false$.

Removal Operation

Removing an element from the Counting Bloom Filter is based around the same logic used in the previous operations, where first a representation of the element to be removed has to be created and then an operation involving the representation and the byte array takes place.

```
Data: $x$ is the object key for the element to be removed.
Function: remove($x$)
    /* Check that element $x$ is member                          */
    if search(x) == true then
        for j : 1...k do
            $i \leftarrow h_j(x)$;
            $B_i' \leftarrow B_i - 1$;
        end
    end
```

Algorithm 5: Counting Bloom Filter remove function

The Algorithm 5 shows the logic behind the operation. Assuming the element x, that is to be removed, belongs to the set, the process that follows takes place. Firstly, k hashes are generated by running element x through the hash function h_j (where $j \in 1...k$). Then, these k hashes are transformed into element x's presentation, which is achieved by translating each of the k hashes to the i positions of the byte array have the element's representation stored. Removing this representation from the byte array is done by setting the new value B_i' to be the old value B_i minus 1, meaning that each i position is decreased by 1.

Then, after successfully removing element x's representation from the byte array, the element itself has been removed from the Counting Bloom Filter, as all knowledge of that element has been erased from the structure.

One problem that can arise with this operation relates to the fact of the possibility of false-positives. There could be cases where elements not belonging to the filter are asked to be removed. If by chance one of the elements falls under the false-positive scenario, all elements that share positions in the byte array with this *ghost* element will be affected, as all of those positions will be decreased by one. Positions that during this process are set to 0 will cause the filter to loose knowledge about members of the set, as their representation wrongly erased when removing the *ghost* element.

By modifying the traditional Bloom Filter, Counting Bloom Filters add support for element removal. Although support for element removal is added, which was one of the original Bloom Filter's limitations, one limitation is still present: the filter can not grow beyond its original length. This problem is the problem covered in the next sections by both scalable versions of the Bloom Filter.

3.1.3 Scalable Bloom Filters

Although the modification previously described remove one of the original limitations of the Bloom Filter — no support for element removal — the problem of having a fixed length filter still remains to be addressed. The objective of the Scalable Bloom Filter is to overcome said limitation, enabling the filter to grow when needed.

The idea behind this structure is to have not one structure that is used to store elements' representations but many. The intuition here is that by having a bucket of Bloom Filters by adding more Bloom Filters to the bucket, more elements can be stored in the structure.

This is achieved by creating an additional sub-filter whenever the filter becomes full, and performing further insert operations in this newly created sub-filter.

$$\rho = 1 - \prod_{i=0}^{l-1}(1 - \rho_0 r^i) \tag{3.2}$$

Another core difference between this filter and the Bloom Filter is the way that the false-probability rate ρ is calculated. In the scalable version of the Bloom Filter, the false-probability rate has to take into account the fact there are many sub-filters, therefore it needs to take into account the false-probability rate of those sub-filters when calculating ρ for the Scalable Bloom Filter. When a new sub-filter is created, in order to maintain the same false-positive for the filter, the false-positive rate for the sub-filter has to be adjusted in order for the false-positive rate of the filter to tend to the desired rate. The Equation 3.2 gives the compounded false-probability rate for the scalable version of the Bloom Filter, where l is the number of sub-filters, r the tightening ratio and ρ_0 the false-probability rate of the first sub-filter [28].

The operations that are supported by this structure are the same as the ones provided by the Bloom Filter interface: i) insertion of an element; and ii) searching for an element. Even though the operations offered by these structures are the same, since the Scalable Bloom Filter entails the presence of a bucket of Bloom Filters, the process behind each of these operations is slightly different.

```
Data: $x$ is the object key to insert into the Scalable Bloom filter
Function: insert($x$)
    $s \leftarrow$ number of sub-filters;
    $p \leftarrow false$;
    /* check if $x$ is already present in any sub-filter */
    for $o$ : $1...s$ do
        if $x \in SF_o$ then
            $p \leftarrow true$
        end
    end
    /* $x$ was not present in any of the sub-filters, and can be
       added to $SF_s$ */
    if $p == false$ then
        for $j$ : $1...k$ do
            $i \leftarrow h_j(x)$;
            $SF_{s'_i} \leftarrow 1$;
        end
        /* Check if the Scalable Bloom Filter needs to be
           expanded */
        if $SF_s$ is full then
            expand();
        end
    end
```

Algorithm 6: Scalable Bloom Filter insert function

Insert Operation

The insert operation is done in three steps, as depicted in Algorithm 6. When inserting element x, the Scalable Bloom Filter first verifies if that element is not already present in any of its sub-filters. This is done by checking for membership of element x at each sub-filter SF_o (where $o \in 1...s$, with s being the number of sub-filters). If after this process element x fails the membership test in all sub-filters, this means that this element is new to the Scalable Bloom Filter, and the operation can go to the next step. However, if any of the sub-filters claim to have element x, the insert operation stops, as there is no need for adding an element that is already in the filter.

The next step takes place at the latest created sub-filter SF_s. The reason for always performing the insertion of new elements in the latest created sub-filter comes from the fact that this sub-filter still has room for more elements, contrarily to what happens in the other sub-filters. Inserting element x in this sub-filter is done by applying the same logic described in the insert operation for the Bloom Filter. Where k hashes are generated by running element x through a hash function h_j (where $j \in 1...k$), and translating the results of this operation into element x's representation. The representation of this element are all of the i positions in the sub-filter's bit array yielded by the previous step. Then, in order to complete the insertion of element x, each i position is accessed and its old value B_i is set to 1 (B'_i).

```
Data: $x$ is the object key for which membership is tested.
Function: search($x$)
    $m \leftarrow false$;
    $s \leftarrow$ number of sub-filters;
    /* check if $x$ belongs to any sub-filter */
    for $o : 0...s$ do
        $SF \leftarrow subfilter(o)$;
        if $x \in SF$ then
            $m \leftarrow true$
        end
    end
    return $m$;
```

Algorithm 7: Scalable Bloom Filter search function

After the completion of the second step, element x has successfully been inserted in sub-filter SF_s, thus being now a member of the Scalable Bloom Filter. The last step is to verify if the filter has to be expanded. The sub-filter SF_s is checked to see if its maximum capacity as been reached. If so, the Scalable Bloom Filter needs to be expanded. This is done by creating an additional sub-filter SF_{s+1} and adding it to the bucket of sub-filters, thus effectively expanding the Scalable Bloom Filter, as more room for new elements is added.

Search Operation

The idea behind the search operation is similar to the one of the Bloom Filter and Counting Bloom Filters where k hashes are generated then translated to i positions, that are then checked to attain if the element belongs or not to the filter. The difference between those operations and the one of the Scalable Bloom Filter relates to the fact that this filter stores the representation of elements in its sub-filters. Hence, the operation — as shown in Algorithm 7 — varies slightly from the how the insert operation is done by the other types of Bloom Filters.

When searching for element x, the Scalable Bloom Filter iterates its sub-filters, and invokes the search operation for element x at each sub-filter. This process follows the logic described in the previous sections, where k hashes are generated by passing element x through a hash function h_j. Then, the result from h_j is used to attain which positions i are used to store the representation of element x. These positions are then checked to verify if all of the values are set to one. If so, element x belongs to that sub-filter, thus it belongs to the Scalable Bloom Filter.

The search for element x ends when it is found in one of the sub-filters, and in this case element x belongs to the filter. Or if element x was not found in any of the sub-filters, and therefore element x does not belong to the filter.

```
Data: $x$ is the object key to insert into the Scalable Counting Bloom filter
Function: insert($x$)
    $s \leftarrow$ number of sub-filters;
    $p \leftarrow false$;
    /* check if $x$ is already present in any sub-filter */
    for $o$ : 0...$s$ do
        if $x \in SF_o$ then
            $p \leftarrow true$
        end
    end
    /* $x$ was not present in any of the sub-filters, and can be
       added to $SF_s$ */
    if $p == false$ then
        for $j$ : 1...$k$ do
            $i \leftarrow h_j(x)$;
            $SF_{s'_i} \leftarrow SF_s + 1$;
        end
        /* Check if the Scalable Counting Bloom Filter needs to
           be expanded */
        if $SF_s$ is full then
            expand();
        end
    end
```
Algorithm 8: Scalable Counting Bloom Filter insert function

3.1.4 Scalable Counting Bloom Filter

Finally, the Scalable Counting Bloom Filter is a structure that combines Scalable Bloom Filters with Counting Bloom Filters, making possible to remove elements from the filter. This is achieved by changing the type of filter used in the sub-filters. Here, Counting Bloom Filters are used, as this type of filter already supports elements to be removed.

The main idea behind this structure is to add support for element removal, while still maintaining support for scalability. The intuition here is the same as the one in the Scalable Bloom Filter. Where, a bucket of sub-filters is used to provide the structure with the scalability property, and then, the element removal is added by using the Counting Bloom Filters for its type of sub-filters.

This Scalable Counting Bloom Filter interface provides the following operations: i) element insertion; ii) element lookup (checking for membership of an element); and iii) element removal. The following paragraphs detail how each of these operations operate.

Insert Operation

The insert operation follows the logic shown in Algorithm 8. The key idea for element insertion, is that the element to be inserted should be inserted in the latest created sub-filter, and have the Scalable Counting Bloom Filter expand after the insertion of the element, in the case that the sub-filter in which the element was inserted become full. This is done in order to make room in the filter for more elements, hence this is the process that adds scalability support.

When insertion an element x, the filter first checks if this element is new, as if it is not, there is no need to insert the element, as it is already present in the filter. This is done by checking for membership of element x in each of the sub-filters SF. The Scalable Counting Bloom Filter iterates its sub-filters SF, and for each SF_o (where $o \in 1...s$, being s the number of sub-filters) the search function in invoked for element x. If after this process no sub-filter claims that element x belongs to its set, then the element is ready to be inserted.

After the process of verifying that element x is, in fact, a new element, the insertion is to take place at the latest created sub-filter SF_s. Adding element x to this sub-filter follows the same process as inserting an element in a Counting Bloom Filter where when inserting an element x, the sub-filter calculates the positions that are to be used to store the element representation, and increments those positions in its array of bits.

The final step that is required for completing the insertion of element x, is to check whether the sub-filter SF_s became full or not. If it is verified that the sub-filter SF_s has reached its maximum capacity, the Scalable Counting Bloom Filter needs to be expanded, as there is no more room left for inserting new elements. The expansion process is done by creating a new sub-filter SF_{s+1} and adding it to the bucket of sub-filters. All further insert operations will be performed in this sub-filter, until a new expansion of the Scalable counting Bloom Filter takes places.

Search Operation

The search operation in the Scalable Counting Bloom Filter is identical to the one of the Counting Bloom Filter, where the membership for an element is performed at each of the sub-filters.

When searching for element x, the filter iterates its sub-filters SF and invokes the search operation for element x at each of them. The search operation performed in a sub-filter is same that was previously described for the Counting Bloom Filter. The search operation stops when a sub-filter reports that element x belongs to it, thus it is present in the filter, or if after iterating all sub-filters the element was not found. This logic is shown in Algorithm 9.

Removal Operation

The removal of an element from the Scalable Counting Bloom Filter is done by using a similar logic to the one applied in the insert operation. Assuming that the element x, that is to be removed, belongs to the Scalable Counting Bloom Filter, the process of removing element x involves iterating the sub-filters and invoking that same operation at each sub-filter. This operation ends when element x is removed from one sub-filter,

```
Data: $x$ is the object key for which membership is tested.
Function: search($x$)
    $m \leftarrow false$;
    $s \leftarrow$ number of sub-filters;
    /* check if $x$ belongs to any sub-filter */
    for $o : 0...s$ do
        if $x \in SF_o$ then
            $m \leftarrow true$
        end
    end
    return $m$;
```

Algorithm 9: Scalable Bloom Filter search function

as allowing the process to continue beyond this point would suggest that elements other than x could be affected due to the false-positive probability of this structure as one of the other sub-filters could also report that element x belongs to them.

When the operation is invoked in the sub-filter, the process described for removing an element in a Counting Bloom Filter takes place. After the process finishes element x has been removed from the sub-filter, thus having been removed from the filter itself.

As stated earlier the implementation of the Scalable Counting Bloom Filter is based on the idea of having a group of sub-filters, and use these filters to perform the operations offered by the interface. This is typically achieved by having a list of Counting Bloom Filters, which grows when the latest sub-filter has become full.

3.2 Summary of Filters

Despite the Scalable Counting Bloom Filter supporting insert, removal, and search operations, it still needs to be adapted in order to be used for state summarization. The fact that the filter is used to summarize the state of the elements that were added, the filter itself is not efficient to be used for state summarization. This is due to the fact that a Scalable Counting Bloom Filter can have many sub-filters, hence the need to create a process that allows for the filter to create a state summarization of itself. This would involve creating a *token* by iterating over all of its sub-filters, and for each of those sub-filters create a state summarization. This operation wastes more resources, the more sub-filters are present. The following sub-section presents the Merkle Tree structure, and how it is used to summarize the state of the Scalable Counting Bloom Filter.

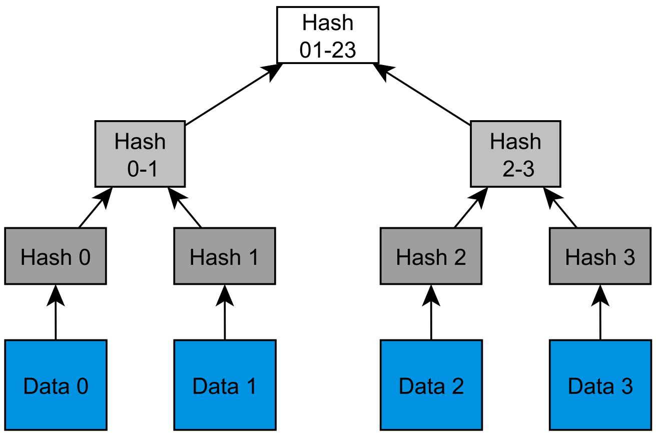


Figure 3.4: Overview of a Merkle Tree

3.2.1 Merkle Tree

Merkle Trees are often used in the context of replicated services [17] as a space-efficient way for state summarization. The main idea behind this structure is to have a summarization of state (*token*) always up-to-date at the root node (see Figure 3.4). By adding new elements strictly to leaf-nodes, and using the remaining nodes as storage for the summarization of state of the nodes bellow, allows for an efficient way of ensuring that the root node has at any given moment in time the up-to-date state summarization *token*.

Insert and Removal Operation

When a new element is added to the tree, a new leaf-node is to be created and used to store the new element. After the leaf-node is added to the tree structure, it invokes its parent node to update its hash, which is done by hashing the combination of the child nodes. After the parent node updates its hash, it invokes its parent so the process can also take place there, and so forth the root-node has been reached. Doing so requires for at maximum h hashes to be recomputed – where h is the height of the tree – for any leaf node that is either modified or added to the tree.

Figure 3.5 exemplifies the aforementioned process, where the block labeled *Data 0* is inserted (or modified), and the path of nodes that need to recompute their hashes in order to reflect the change to the objects in the Merkle Tree. In this case, the block *Data 0* is modified and the path followed goes from the node *Hash 0* — which has the hash for *Data 0* — to the root-node, which is the node given that the modified block was *Data 0*, the path followed goes from the node *Hash 0* to the root-node, passing through the node *Hash 01-23*. By calculating the new hashes for all of the nodes in this path, the Merkle Tree is able to correctly reflect the modification that took place, where a new node was inserted in the tree. This example also shows how efficient this structure is, as only h nodes had to recompute their hashes to successfully reflect the new inserted node — where h is the

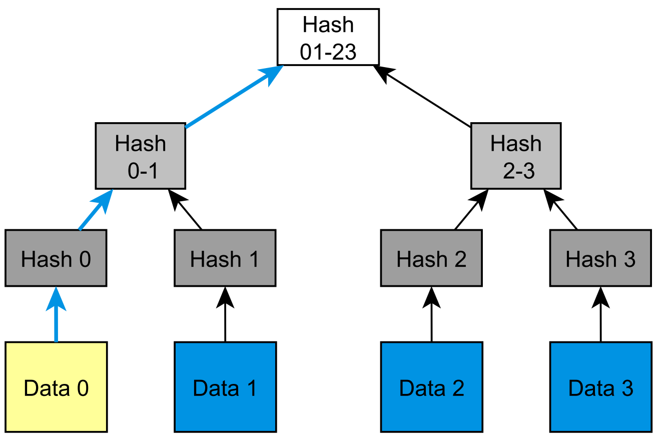


Figure 3.5: Merkle Tree insert operation

height of the tree.

Deleting an element from the tree follows the same logic. Firstly the node containing the element to be removed is located and removed, and then the previously described process for computing the hash of the tree takes places.

3.2.2 Scalable Counting Bloom Filter Summarization

One of the problems in the Scalable Counting Bloom Filter is that it is not very efficient at summarizing its state, as all sub-filter have to be iterated in order to generate a state summarization *token*. An alternative approach to summarize the Scalable Counting Bloom Filter is to take advantage of the efficient way that the Merkle Tree has of generating a the state summarization *token*.

The idea behind merging both structures together is to have the sub-filters of the Scalable Counting Bloom Filter as nodes in a Merkle Tree. Thus, providing the Scalable Counting Bloom Filter with an efficient mechanism to summarize its state. By having the sub-filters in a Merkle Tree, modifications that take place at a sub-filter can efficiently be reflected by the Merkle Tree, as only h hashes have to be recomputed to generate an up-to-date state summarization *token*.

When a sub-filter is modified the aforementioned process for calculating the root hash in the Merkle Tree takes place, as shown in Figure 3.6. In this figure we can see that element x is inserted in the sub-filter *Sub-Filter 3*, and that all the nodes that belong to the path going from *Sub-Filter 3* to the node *token* are involved in the process of recomputing the hash. That is, element x is inserted in the Scalable Counting Bloom Filter, which implies that x is inserted in the latest created sub-filter — in this case *Sub-Filter 3* — which is stored by *Node 4*. This node then recomputes its hash, and has its parent *Hash 2-3* do the same. Finally, the root-node is reached, and it also recomputes its hash, successfully reflecting the change that took place at the leaf-node *Node 4*, thus generating an up-to-date state summarization *token*.

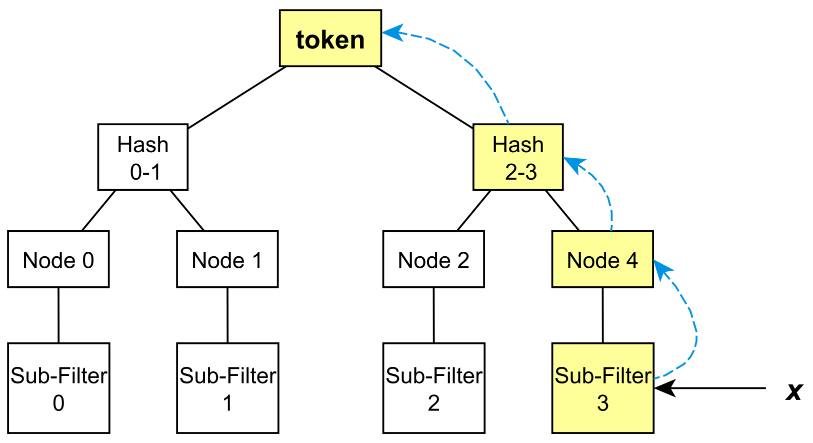


Figure 3.6: State Summarization for a Scalable Counting Bloom Filter using a Merkle Tree.

This process is valid for any type of modification that the Scalable Counting Bloom Filter suffers, whether it be a element that is inserted/modified or removed, the state summarization *token* is generated in the same way, due to the fact any modification in the filter triggers the process of updating the *token* in the Merkle Tree.

4

Summarizing Replicated Service State

State summarization is usually achieved by building a Merkle Tree over the state of the service. Using this summaries to compare replicas in a replicated services requires that the same Merkle Tree is built when replicas are in the same state. For services replicated using the state-machine model, it is reasonable to assume that the internal state of the service will be the same, when ignoring pointers and considering state only. This might be no longer true if the service executes operations in different order, which may lead the internal service to differ in different replicas. The same happens if different replica implementations are used. To address this issue, some replicated services build Merkle Trees over the abstract state of the service, which must be exported. This is rather complex and may lead to replicas generating different state summarization *tokens* even if they have executed the same exact operations, as they are free to choose the order of execution for the operations.

For those replicated services that ensure that replicas use a sequential order of execution, the previously described techniques for state summarization can be used, as they ensure that a replicas that have not diverged from the others during operation will have identical state summarization *tokens*.

In this scenario the state of the replica can be summarized by using a Merkle Tree, which is an efficient structure for generating a state summarization *token*. When a replica executes some operation, the objects that compose the internal state of the replica can be updated in the Merkle Tree, triggering the state summarization process. In this process the Merkle Tree computes the new hashes from the node that was affected to the root node, thus updating the state summarization *token* that is stored at the root of the tree. The problem with using this structure to achieve state summarization is that the resulting summarization of state is affected by how the tree itself evolves, the order by which the objects are added to the tree is also reflected by the summarization. This problem is solved in the Scalable Counting Bloom Filter by having the summarization of state being independent of the structure internal representation, in this structure this process reflects only the the objects that are inserted/removed, and not of the underlying internal structures of the Scalable Counting Bloom Filter.

Likewise, the Scalable Counting Bloom Filter could also be used in this scenario. When a replica finishes executing an operation, the object modified by the operation in the internal state need to be update in the filter. Here a distinction between three possible types of modifications has to be made. If the object is removed from the replica, for example, a database that erases a table row, the delete operation would be called at the Scalable Counting Bloom Filter. The opposite logic would be used for a new object that is added to the replica, where it would be added to the Scalable Counting Bloom Filter. For update operations — an object that is already present in the replica that gets modified — the original object has to be also used by the filter. This is done in order to assure that the state summarization *token* generated correctly reflects the internal state of the replica. In this case the original object is first removed from the filter — by calling the remove operation for that object — and only then the modified version of the object is inserted in the filter.

After the aforementioned process for updating the objects in the Scalable Counting Bloom Filter has been completed, the filter generates a state summarization *token*. This *token* is generated by iterating the Scalable Counting Bloom Filter sub-filters, and for each sub-filter an hash is generated. This hash corresponds to the summarization of state of the sub-filter, and it is typically generated by hashing the contents of the array of byte used by the sub-filter to keep track of the added elements. Then, all of the hashes are combined, and hashed one more time. This last hashing process effectively generates an hash of the sub-filters' hashes, thus creating a hash that is used by the Scalable Counting Bloom Filter as the *token* that summarizes its state.

4.1 Summarization in the presence of non-sequential operations

The purpose of the previously described structures — Merkle Trees and Scalable Counting Bloom Filters — is to be used as an efficient way of representing the internal state of a replica, by generating a *token* that reflects the elements encompassed in the structure.

Although these structures are able to achieve their goals in scenarios where the order by which the elements are added is the same for all replicas, the same can not be guaranteed when the order is not the same, albeit the need for an alternative solution to provide a way for these structures to be used in a context where operations are non-sequential.

This results in these structures not being able to correctly summarize the state of the service. Independently of the structure used, if replicas at the service have to execute operation o_1 ... o_n, if the order by which these operations are executed at the replicas varies, the state summarization *token* may differ from replica to replica. As the operations are reflected in the structures at different moments in time, and despite the fact that all operations will eventually be reflected, the end result can be different

In the case of the Scalable Counting Bloom Filter, the aforementioned problem derives from the fact that elements added in different orders may end up being stored at different sub-filters. For example, if we consider a replicated service that has two replicas, and is to execute operation o_1 and o_2. If the Scalable Counting bloom Filter at each of those replicas is at verge of being expanded — they only support one more operation before needing to expand — by executing the operations in a different order, this means that the operations will be reflected in different sub-filters at the replicas. If replica r_a executes operation o_1 and then o_2, by having only room in its current version of the filter for one operation, before triggering the expansion of the filter, this signifies that the resulting state from operation o_1 is added to sub-filter sf_i and the resulting state from operation o_2 to sub-filter $sfi + 1$.

The opposite happens for the second replica r_b, where operation o_2 executed first than operation o_1. In this case the resulting state of operation o_2 is added to sub-filter sf_i, whereas the resulting state of operation o_1 is added to sf_{i+1}. This results in a both replicas generating a different state summarization *token*, despite both having executed the exact same operations.

Likewise, for Merkle Trees if replicas execute operations in a non-sequential way, thus elements are added in different order to the trees, this results in the possibility of having divergent state summarization *tokens*, despite having execute the same operations. This problem is related with how the tree is internally represented, if the Merkle Tree is implemented by using a self-balancing tree, which typically perform some kind of node rotation when elements are added, the insertion order will affect the summarization of state, as different orders will trigger different rotations.

If we translate the aforementioned example to the context of Merkle Trees, where two replicas — r_a and r_b — execute operations o_1 and o_2 in a different order the former replica will store operation o_1 at, for example, node n_1, and operation o_2 at node n_2. Whereas the later replica will store operation o_2 in node n_1, and operation o_1 in node n_2. After calculating the new state summarization *token*, by following the path that goes from the new or modified nodes to the root-node of the tree, given that this path is different at replica r_a from replica r_b, the *token* that is generated is necessarily different too.

The aforementioned scenarios pose a challenge, as both of these structures can not be directly integrated in a replicated system's replica to provide a way for state summarization in the presence of non-sequential operations. The goal for the remaining of this section is to explain how the Scalable Counting Bloom Filter can be modified in order to fit the context of summarizing the state of a replicated service that does not enforce sequential order of execution, more specifically it will be detailed how this is achieved in the context of a replicated database service.

The challenge posed by non-sequential operations still stands in the context of replicated database services. This can be solved by enforcing mechanisms at the replication service itself, which purpose is to ensure that the aforementioned structures can be used to summarize the internal state of a replica.

The idea is to have the replicated service allocate operations in blocks, ensuring that these operations are exactly the same for blocks of operations sent to any given replica. By following this approach we create a sense of a sequence of execution. The fact that even though these blocks of operations contain the same operations, they can still be execute in different order at the replicas. This second problem is solved by modifying the original behavior of the Scalable Counting Bloom Filter, restricting the filter from expanding in the middle of a block of operations.

The intuition here is that by buffering a certain number of operations before these are reflected in the state summarization structure — Scalable Counting Bloom Filter — it will be possible to generate the same exact *token* despite the order by which the operations took place was different. The buffering process requires for the same exact operations to be buffered — the order by which this occurs does not matter. It is the responsibility of the replicated service to ensure this. For example, in the case of a replicated database service, this can be achieved by ensuring that a certain number of transactions is performed at the replicas before requesting new operations to be executed. Given that in this case it is expected from the service to be always executing transactions, only a small dent in the performance aspect takes place. As the time between a buffered block of transactions being reflected by the filter, and a new one being formed should be small enough for the performance hit be negligible.

By scheduling a certain number of transactions to take place before another block comes in, the Scalable Counting Bloom Filter can be used as a state summarization mechanism as it will yield deterministic *tokens* for replicas that do not diverge during operation. One problem that can be pointed to this solution is that during the buffering process some transactions can result in a replica diverging from the others, and the replicated service only being able to identify this replica's incorrect state after the completion of the execution of the block of transactions. This means that there will be resources being wasted — executing further transactions — when the replica has already diverged.

Recalling the aforementioned example, where two replicas had two operations to execute, and by executing these operations in different order they generated different state summarization *tokens*, due to the fact that the filter expanded at different operations in both replicas. Where operation o_1 triggered the expansion of the filter in replica r_a, and operation o_2 triggered replica r_a's filter to expand. In principle, by restricting the moments where replicas are allowed to expand the filters, the problem should be solved. Given that the replicated service already ensures that a block of operations contains the exact same operations, by restricting the moments where a filter is allowed to expand, the resulting *token* should be exactly the same. The idea here is that by only allowing a filter to expand after a block of operations is reflected in the filter, the exact same operations are added to the same sub-filter at any of the replica's Scalable Counting Bloom Filters, despite the order by which the operations in a block are executed.

The reason behind choosing to use the Scalable Counting Bloom Filter over the Merkle Tree to represent the state of the replica is that the filter grows deterministically, in the sense that the order by which elements are added does not influence the resulting *token*. Merkle Trees are typically implemented on top of a self-balancing tree (e.g., Red-Black Tree), and this restricts their applicability for summarizing the state of a replica in the presence of non-sequential operations, resulting in replicas that execute the same exact operations in different orders yielding different state summarization *tokens*, as the order by which the resulting states from those operations are added to the tree will trigger distinct rotations of the tree's nodes.

By combining both approaches – Merkle Trees and Scalable Counting Bloom Filters – an efficient and accurate way of summarizing the internal state of the replica is created. Not only does the resulting structure grow deterministically throughout execution, but it also ensures that the generation of the state summarization *token* occurs efficiently, as the integration of the Merkle Tree in the Scalable Counting Bloom Filter guarantees that the number of hashes that need to be computed is reduced.

5

State Recovery

The previous section explained the process behind generating a *token* that summarizes the state of a replicate service's replica. In this section it is described how that same *token* triggers the process of recovering a faulty-replica, and how the recovery takes place.

Performing the recovery of a replica first requires the replicated service to be able to identify those replicas that have become faulty, and therefore need to be recovered. This can be accomplished by applying the aforementioned concept of *token* described in previous sections. The replicated service can then make use of this *token* in order to identify replicas that have diverged during operation, by comparing the *tokens* that are generated by the replicas throughout their execution.

The following explanation assumes that the replicated service uses enough replicas, that allow for faulty-replicas to be identified by making use of the aforementioned *token*. That is, the number of replicas used allows for a consensus of what the correct value for *token* should be after operations are executed. If this is not guaranteed, a replicated service that makes use of only two replicas is not able to identify a faulty replica when their *tokens* differ, as one of the following explanations could be given to justify the different *tokens* generated by the replicas: i) both replicas have failed; ii) only one failed, but there is not enough information to determine which of the replicas failed. Yet another problem present itself in this scenario, even if both replicas generate the same *token*, they may also have both failed, and by not be able to identify this, the replicated service will be incorrect until the end of execution, as more operations are executed, the more the error propagates.

Algorithm 10 depicts how the replicated service identifies replicas that diverged during operation, thus have become faulty. Here, r replicas are to execute operation x. After all replicas execute x, a consensus algorithm is run. This allows the replicated service

```
$r \leftarrow$ Number of Replicas;
/* Execute operation x                                        */
for $i \in \{1, ..., r\}$ do
    $Replica_i$.execute(x);
end
/* Determine the majority token                               */
Token $\leftarrow$ getMajorityToken();
/* Identify replicas that have diverged                       */
for $i \in \{1, ..., r\}$ do
    /* If the token of a replica differs from the majority
       token, the replica is deemed to faulty                 */
    if $Replica_i$.getToken() $\neq$ Token then
        $Replica_i \leftarrow$ faulty ;
    end
end
```

Algorithm 10: Process for identifying faulty-replicas

to determine which is the correct *token*, as it assumes that if the majority of the replicas generated the same *token*, that *token* reflects the state of a correct replica. Then, the *tokens* from every replica is compared to the majority *token*, if a replica's *token* differs from this, it is marked as having diverged, and its state is considered to be incorrect, hence the replica itself if inconsistent.

Taking for granted that the replicated service uses enough replicas that allow for faulty-replicas to be identified, the next step is to recover those replicas. After having identified a replica that has become faulty, the service needs to trigger the recovering procedure.

The main idea behind the recovery procedure is that after a replica undergoes the recovering procedure, the result from executing the recovery is that the replica is healed, and can continue its normal operation. Since the basis for identifying a faulty-replica is the *token*, the basis for recovering a replica is intrinsically related with the *state summarization* process. Hence, recovering a replica is done by recovering the elements that are used for the creation of the *token*.

There are several techniques that can be employed to recover a replica, differing in the way they achieve their goals according to the type of replication techniques used by the service. The following sections describe how the state recovery mechanism is implemented for the different types of replication techniques.

5.1 State Recovery for Speculative Replicated Services

As explained in Section 2.2.3, *speculative execution* is a technique used to improve the overall performance of a replicated service, by preemptively executing operations that are dependent on the results of a previous operation without waiting for the arrival of those results, thus the service speculates the result from that operation. This speculated result is then used by the replica to execute further operations, without needing to wait for the real result of the operation to arrive.

This allows the replicated service to improve its performance, as the contention at the replicas is reduced, by not having replica sit idle waiting for results of previous operations to continue the execution of new operations, thus gaining performance.

The problem with this type of replication technique is the fact that if a speculated result does not match the real result, the state of the replica is affected, as the replica diverges in state by having used an erroneous speculative result to execute other operations.

For example, if the service speculatively executes operation o_2 that depends of the result of operation o_1, and o_1 fails to execute properly, o_2 will also be affected. As the execution of o_2 took place on the basis that the speculative result was correct, thus by wrongly calculating this value to execute further operations — or simply by having diverged during o_1 — the replica state becomes faulty.

Traditionally, services based on speculative execution address this problem by creating *checkpoints* throughout the replica's lifetime. This allows the replica to have a fall-back point in time where its state was know to be correct, which enables the service to revert the replica to previously known correct state in the case of a wrong speculative result. These *checkpoints* are created after the replicated service confirms that the result from the speculative execution was correct, which translates in the state of the replica being correct. This verification is done by confirming that the speculated result, used by the replica to execute further operations, matches the real result.

This *checkpoint* is then to be used by the replicated service to revert the replica to a correct but out-of-date state if the replica diverges during operation. Divergences between replicas' states is done by applying any of the available state summarization techniques, and comparing that the summarization of state is identical across all replicas.

When the service identifies that a replica has diverged during operation, possibly due to a wrongly speculated result, the replica uses the last available *checkpoint* to correct its state. At this point, the replica's state is deemed to be correct but out-of-date. The next step in the state recovery process is to bring this replica's state up-to-date. This is done by executing the operations, that lead to a divergence in state, using the real result.

Although this approach effectively avoids running into another divergence related to another wrong speculative result, it does not ensure that the replica's state is correct, as the replica might still diverge due to fail-stop/self-evident errors. If the replica's state fails again to converge with the state of the remaining replicas, an alternative approach for recovering its state is performed. This is done by applying the pro-active recovery technique, where the replica is restarted and rebooted, which ensures that the replica's state is blank. Then, the replica is brought up-to-date by transferring the state of a correct replica, replacing its blank state, thus effectively recovering the replica.

In the above example, if the replica had created a *checkpoint* c_0 before o_1, the recovery process for that service would *rollback* that replica's state to that point in time (c_0), as the replica was known to be in a correct state, and discard all the operations that took place after o_1. Then, the replica's state can be brought up-to-date by replacing its state with the state from a replica that has successfully completed the operations.

The reason behind the existence of the first method for recovering a replica's state — reverting the replica to a previously known correct state and then re-executing in a non-speculative manner the block of operations that are pending — is due to the fact that if this approach successfully recovers the state of the replica, less impact is felt on the replicated service.

Given that a replicated services that applies speculative execution does so to reduce contention among replica, thus improving performance, it will favor a method for recovering replica's state that has less impact — performance-wise — in its normal operation, even if this means that in some cases two different forms of state recovery are to be executed to recover a fault-replica. The second approach — pro-active recovery — as a greater impact the service's performance as the transference of state suggests that a correct replica would have to waste resources exporting its state, before the faulty-replica was successfully recovered.

5.2 State Recovery for Design Diversity-Based Replicated Services

There are two different types of *design diversity*: i) diverse; and ii) non-diverse. Although the recovery of state in a service that applies *diverse design diversity* use the same logic behind one that applies *non-diverse design diversity*, some implementation details differ, thus this type of diversity can be seen as a special case.

We start by describing the process of recovering a replica for a service that applies *non-diverse design diversity* and then detail the changes that need occur in order for the state recovery mechanism to be applicable in the context of a replicated service that applies *diverse design diversity*.

Non-Diverse Design Diversity

In this context, given that all replicas present in the replicated service are identical, meaning that they all run the same software implementation, recovering a faulty-replica is done by transferring the state of a correct replica to the faulty-replica, thus having its state coverage with the remaining replicas.

This requires for the presence of a state transference mechanism, which entail that the replicas are to support the following operations: i) export their state; and ii) import the state from another replica.

Exporting the state of a replica can be achieved by gathering the objects used by the state summarization process to a general *token* that represents the current state of a replica, and exporting said objects. The exportation of these objects can take various forms: i) it may be a serialized version of the objects; or ii) the objects themselves. Depending on the type of replicated service, opting for one approach over the other may yield better results.

After the exportation of state completes, the result has to be transfered to the faulty-replica in order for it to be recovered. At this point, after the recovering replica receives this exported state, is replaces the correspondent objects with the ones from the imported state. When the recovering replica finishes this process, its state is now considered to be correct, as the objects used to create the state summarization *token* have all been replaced with the ones from a known correct replicas, thus the replica is considered to have terminated its recovery, and is now deemed to be correct.

Diverse Design Diversity

Although the aforementioned process works for the majority of cases, in order for it to be applicable in the context of a replicated service that uses *diverse design diversity*, changes have to be made to the mechanisms involved in the process of exporting and importing states.

The challenge of recovering the state in this context comes from the fact that replicas are not all running the same software implementation. Despite the implementation running in each replica obeying to the same software specification — independently of the way that the software is implemented, given the same inputs the outputs yielded are to be the same — they differ in the way that they are implemented.

If before exporting and importing the state was achieved by exporting and importing the objects used by the state summarization process to summarize the state of a replica, given that for this type of *design diversity* different replicas can be running different software implementations, there is a strong chance that the objects used in each implementation vary, thus the exportation and importation of state has to be adapted to enable replicas running different implementations to be capable of importing and exporting states amongst themselves.

This requires that in addition to the functions required for recovering the state in the *non-diverse design diversity* scenario, for functions that translate implementation specific states to an abstract state and *vice-versa*, which will enable the recovery of replicas running different implementations.

For example, if we consider a replicated database service that runs two different databases implementations: I_a and I_b. If a replica running implementation I_a becomes faulty, and it is to be recovered with aid from implementation I_b, one possible abstract state that I_b could export is the contents of the database itself. This would work as the contents of the database are independent from the objects used by I_a and I_b to store these contents.

6 Prototype: implementation and evaluation

This section explains how the aforementioned concepts and techniques for summarization of state and state recovery were applied to both *H2 Database* and *HSQL Database*. We explain how the state summarization and state recovery for both databases can be used by a replicated service to improve its resilience to failures. We also perform a study of the overhead added by the aforementioned mechanisms to the databases by comparing the time it takes to execute a certain amount of operations in both modified (with the Scalable Counting Bloom Filter integrated) and un-modified versions of the databases.

6.1 Implementation

Here it is explained how the concepts and techniques described throughout the document were applied to *H2 Database* and *HSQL Database*.

Independently of the type of technique used by a replicated service, in order to improve its resilience to failures it needs to incorporate to major components: i) state summarization; and ii) state recovery. This means that the replicas of the replicated service need to integrate the aforementioned components to provide the service with a way to identify faulty-replicas and recover said replicas.

In the context of this work support for state summarization and state recovery are added to the *H2 Database* and *HSQL Database*. The following sections describe the modifications that took place to achieve these goals, and what entails to use these implemented mechanisms by the replicated service.

6.1.1 State Summarization

State summarization is what allows the replicated service to identify replicas that have diverged during operation, thus their state has become faulty and the replica needs to be recovered.

The process of summarizing the state of a replica is done by applying the techniques described in Chapter 3. By integrating a Scalable Bloom Filter, the state of the replica can be summarized. But, as explained in earlier sections, not in all cases it is possible to directly apply this structure, as the end result would not be the desired one. The main reason for this is that many replicated services do not enforce a strict order of execution, which entails that despite replicas having the same operations to execute, they are free to do so in any order they see fit.

Chapter 4 provides a more elaborated way of achieving state summarization, that is also applicable in scenarios in which the replica deals with un-ordered operations, which is the basis for the solution that follows.

The first step in providing support for state summarization for any replica, is to first analyze the replica and identify what composes the state of the replica, which requires identifying the components of the replica that will be used to create its abstract state.

Given that the focus of this work is a replicated database service, and that support for *design diversity* is also an objective, identifying what composes the state of a *H2 Database* replica or a *HSQL Database* replica is not enough. Since the goal is to have replicas running different database implementations — the ones mentioned before — state summarization has to be a compromise between identifying what composes the state of a specific type of replica, and what the each type of replica has in common.

The state summarization begins by deciding what is the state that needs to be summarized. In a database service, the state consists of the data contents —- the internal structures of the database can be left out of state summarization, as it would be possible to locally verify if they are correct from the database contents. Thus, the contents of the tables of the database are used to compose a summary of the replica's state, as the rows of the tables are present in both *H2 Database* and *HSQL Database*.

This approach works despite the fact that the databases are implemented differently, because the values used to create the summary are the same for both implementations. By using the values of the operations — before they are converted to implementation specific objects — it is guaranteed that the values used in the filter for both databases are identical, therefor ensuring that the *tokens* generated through the replica's life-cycle will be the same in the absence of failures.

Next follows a explanation of how this was achieved for both *H2 Database* and *HSQL Database*

As aforementioned, the state summarization *token* is created by using the contents of the databases, more concretely, the values of each column of each rows that gets created in a table. This means that the state summarization process needs to occur at the moment the database interacts with a table. When a transaction arrives at the database, whether its operations are inserts, deletes or updates, the filter needs to be able to correctly reflect the internal states of the databases after the operations are executed.

This is achieved by modifying the database's original behavior, by hijacking the session that is processing the transaction, and having it execute extra operations related to the state summarization process.

The idea behind the implementation of the state summarization for a database is to have the generation of the *token* to be incremental, and to be done alongside the operations performed at the database. Although the data that represents the database is stored in tables, the constructions of the state summarization *token* is to take place without accessing the database's tables, and instead occur when a table gets modified.

The Scalable Counting Bloom Filter is chosen as the state summarization structure to be used to create the former *token*. Given that the operations that arrive at the database are for the most part treated by a session class, and so, insert, delete, and update operations pass through that class, the filter is to be updated at the moment that the session receives an operation.

In this context, the Scalable Counting Bloom Filter is used to summarize the state of a particular table, meaning that a database that has n tables will also have n filters, which means that a more general state summarization *token* has to be created with the values from each table's filter. The reason for having a filter for each tables is the fact that the filters will be smaller, therefore the impact of generating representations for objects will be lower, as less hashes will need to be generated.

Here it is assumed that the replicated service has control over the operations that are sent to the replicas. This means that the service is required to send the same exact operations to all replicas, and that new operations are only sent after the previous block of operations has been successfully executed by the replicas, which can be verified by comparing the state summarization *token* after a replica executes a block of operations. The reason for this requirement is to guarantee that the operations that are used to compose the state summarization *token* are the same for all replicas, independently of the order by which the replicas decide to execute the operations in a block. By assuming that the replicated service sends the same blocks of operations to all replicas, and that new blocks of operations are only sent when the previous operations in a block have been successfully executed, we can guarantee that the state summarization *token* generated by any replica is the same in the absence of failures.

When a transaction is committed at the database, the system goes through the process illustrated in Algorithm 11. The transaction Tx_i is added to a buffer of transactions T_b that are pending to be committed at the filter. This buffer is a list of lists of triplets. Each operation in transaction Tx_i is added to the buffer in the form of a triplet $< T, V, O >$. Where T is the table involved in the operation, which will later be used to access that table's Scalable Counting Bloom Filter. V is the concatenated values of the operation — e.g., for an operation that involved inserting the row A, B, C, the value V would be

```
$T_{id} \leftarrow$ Transaction Identifier;
/* Mark Transaction $T_{id}$ as committed by the Database */
$C_{T_{id}} \leftarrow 1$;
/* Add Transaction $T_{id}$ to the buffer */
$B$.add($T_{id}$) ;
$B_s \leftarrow$ Buffer Size;
$ready \leftarrow$ true;
/* Verify that all the transactions in a block have been
   committed */
while $t \in \{1, ..., B_s\}$ and $ready == true$ do
    if $C_t \neq 1$ then
        $ready \leftarrow$ false;
    end
    $t++$;
end
if $ready == true$ then
    for $x \in \{1, ..., B_s\}$ do
        $t \leftarrow B$.get($x$) ;
        filter.add($t$) ;
    end
    /* Rotate $C$ */
    $C$.rotate() ;
    /* Update Database's state summarization token */
    Database.computeHash() ;
end
```

Algorithm 11: State Summarization Process

ABC — value that should be used in the filter. And O is the type of operation that was performed in the transaction, which is to be the operation performed at the filter.

After adding the necessary information to the buffer, the transaction Tx_i is marked as have been committed by the database in a *BitSet* BS_{ct}. Using a *BitSet* for keeping track of the transaction that have been committed by the database but that have not yet taken part in the database's state summarization *token*, allows for an efficient way for checking which transactions have yet to be reflected by the state summarization structure.

The next step is to verify — after the transaction is executed by the replica — if the latest block of buffered transactions — those that are waiting to be inserted in the filter — have all been marked as committed. If so, the buffer T_b is accessed, and a list of triplets retrieved for each of the transactions that were pending to be added to the filter. Then, for each triplet in a list, the Scalable Counting Bloom Filter of the table T is accessed, and an operation of type O is performed with the value V. This process ends when all of the operations have been reflected in their respective filters. At this point, the transactions have successfully been added to the filters, therefore the buffer can discard them. Additionally, the *BitSet* BS_{ct} is rotated, effectively discarding the information regarding those transactions, which restricts it size from going indefinitely. This is done by creating a new *BitSet* that contains all the values from the old *BitSet* except those that correspond to the last block of transactions that where already reflected in the Scalable Counting Bloom Filter.

At this stage the summarization of state in nearly completed, all the filters of the modified tables have now recomputed the *tokens* that summarize their state, and the only operation that still needs to be made is for the general database's *token* to be generated. This *token* is generated by concatenating all of the *tokens* of the database's tables, and hashing that value. For each table T in the database, the Scalable Counting Bloom Filter of table T is accessed, its *token* retrieved, and concatenated with the remaining *tokens*. After iterating the tables, the concatenated value is passed through a hash function, that generates the state summarization *token* for the database itself.

Although the state summarization *token* has been created, it still needs to be accessible from the outside — replicated service — in order to be useful. As it is this same *token* that allows the replicated service, more concretely its fault-tolerance mechanism, to identify if a replica has diverged in state, therefore its faulty, and needs to undergo the recovering procedure.

This problem was approached by analyzing the interfaces for both databases, and identifying those functions that could be used to retrieve the state summarization *token*. More specifically, our attention was drawn to the functions that provided access to the meta-data of the databases. The intuition here is that some of those functions would have to access variables in the database in order to return some value, which opened the possibility of modifying the behavior of one of the functions to access the *token*, returning it to the outside. Another advantage of this approach is that by using the functions offered by the *JDBC* interface, it was guaranteed that it would be offered by both databases, avoiding the need to create a function in the replicated service to deal with the *token* retrieval.

One function that is offered by the *JDBC* interface, therefore being present for both *H2 database* and *HSQL database*, is a function that allows to access the version of the database. Not only modifying this function does not change the normal behavior of the database, as it is not used in the context of this work, but it also fits the concept of version in a replicated service, as the version of a replica in this context can be seen as its current state, which is given by the state summarization *token*.

6.1.2 State Recovery

State recovery is the process of recovering a replica that the replicated service has identified as being faulty.

The first step is for the replicated service to determine if a replica's state has become inconsistent during operation, thus having become faulty, and needing to undergo the recovery process. The service achieves this by comparing the *token* that the replicas generate after a certain amount of operations take place. This amount of operations corresponds to the size of the transaction buffer T_b mentioned in the previous section, as this allows for a more immediate identification of divergences betweens replicas. By checking the state of the replicas at the exact moment that a transaction block is reflect in the respective table filters, thus checking the *token* for the database state after this has been updated, allows for divergences between replicas to be promptly identified.

After the service identifies a replica that needs to be recovered, the state recovery process is triggered. Recovering the state of a replica in the context of a replicated database service, where the replicas are databases — in this case, *H2 databases* and *HSQL databases* — consists in obtaining the state of a correct replica, and importing said state in the recovering replica.

Given that the state of the replica includes Scalable Counting Bloom Filters for each of the database's tables, and that the *token* that is generated by the filter is based on the contents of the tables, doing a full recovery of a replica can be divided in the following operations: i) exporting the filter from a correct replica; ii) dumping all of the tables' contents of the aforementioned replica; iii) importing the contents of the tables dumped; and iv) importing the previously exported filter in the recovering replica. Note that it is necessary to import the filter from the correct replica, as re-creating an exactly equal filter would require adding the information in the same order.

Exporting the filter

The filter exportation process takes place at a correct replica, ensuring that the recovering replica that will use this filter recreates its filter correctly, thus recovering successfully. This operation is divided in two parts: i) filter serialization; and ii) having the filter be accessible from the outside-world.

The first part — filter serialization — is done by iterating all of a correct replica's tables, and for each of those tables invoke the filter exportation method. What this filter exportation method does, is to serialize the Scalable Counting Bloom Filter in a *JSON* object. This *JSON* object contains not only the variables of the filter that are needed to reinitialize an identical Scalable Counting Bloom Filter, and all of the sub-filters and their own variables.

Additionally, all of the resulting *JSON* objects that are created — one for each table of the database — are paired with their corresponding table, generating a pair $< T, F >$, where T is the table's name, and F is the *JSON* object that represents the Scalable Counting Bloom Filter of table T, after it has been converted to a string of text.

Figure 6.1 depicts the general structure of the aforementioned *JSON* object, after having been converted to a string of text.

The second step — making the serialized version of the filter accessible form the outside — is achieved by using the string version of the *JSON* object that resulted from the first step, and inserting it a table T_f, which is by default accessible from the outside, given that it is a typical database table.

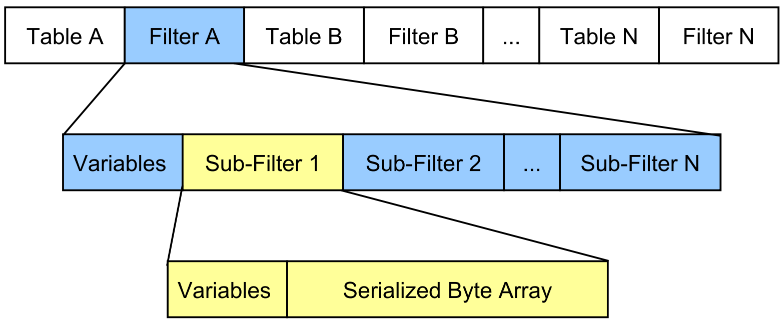


Figure 6.1: Serialized Scalable Counting Bloom Filter's Format

Although performing the aforementioned steps successfully achieved the filter exportation, one problem still needs to be addressed, as it is necessary to identify when is the filter exportation to take place. If the up-to-date serialized version of the filter is to be kept at all times in table T_f, this means that the database is wasting resources by having to update this table each time the transaction block gets full, and the transactions are reflected in their corresponding filters. This would waste resources, as it is expected that a divergence between the state of the replicas is a rare occurrence, and that the recovery process composes a negligible amount of the operations taking place at the replicated service. And by having an always up-to-date serialization version of the filter in table T_F, this would imply that the time taken from the normal operation at the databases to compute this serialization would be wasteful.

The intuition behind the implemented solution for exporting the filter, is to only perform this operation when strictly necessary. It is only to take place when the replicated service identifies that a replica has diverged in state, thus having become faulty, and it needs to undergo the recovering procedure. This implies that it is the responsibility of the replicated service to trigger the filter exportation process, which means that this process is to be triggered from the outside and not from within the database itself. Hence, the filter exportation needs to be triggered by an external event. Since the filter is only needed by the service when a replica needs to be recovered, by having the service perform a select operation in a filter table T_f of a correct replica, the filter exportation is externally initiated.

Another aspect of filter exportation that needs to be taken into account is the fact that throughout a replica's normal operation, transactions are not immediately reflected in the filter. Since transactions are buffered in blocks of size n, allowing the filter to be exported at any moment in time would mean that possibly at most n transactions might be pending to be inserted in the filter when the exportation is taking place. By allowing this to happen, the serialized filter when used to recover a fault-replica, would restrict the applicability of the solution, as is this case, the recovering replica would not be able to successfully converge in state with the remaining replicas, as those pending operations would not be known to the recovering replica.

This problem is solved by restricting the moments that a replica can export its filter. Assuming that the replicated database service is constantly performing transactions, and therefore the replica is constantly executing operations, the exportation of the filter can be delayed until there are no transactions pending to be added to the filter. Or, in the case where the replicated service is not expected to have to deal with constant operations, *dummy* transactions could be created, forcing the buffer of transactions to get full, thus forcing the pending transactions to get added to their respective filters before the exportation process occurs.

Re-populating the tables of a recovering replica

Re-population of the database's tables is done by dumping all of the contents of the tables of a correct replica, and using this dump to populate the recovering replica's database.

For the case of a replica running the *H2 Database*, the data in the tables is dumped to a file by using sql-native commands. The *SCRIPT* command offers a way for the database to create a file with the contents of the database. This file contains not only the necessary commands needed to re-create the tables, but also the contents of each table. The data for each table is represented for each row of a table as a sql insert statement insert sql commands that allow for the recreation of the tables and recreation of their contents.

For the other replica type — *HSQL Database* — used by the replicated service, the process of dumping the contents of the database is done by executing a *Java* program that comes with the database. This program also provides the ability of exporting the contents of a *HSQL Database* to a file. The structure of this file is identical to the file created by the *H2 database*, being composed by sql-commands that create the tables and the contents of the tables that were present at the moment the export took place.

One problem that derives from both databases having different ways of exporting their contents to a file, is that the inverse process — importing contents from a file — is also different. This means that there is no direct way of using a file created by ah *H2 Database* in an *HSQL Database*, and *vice-versa*. This problem can be solved by creating abstract functions, whose job is to convert from a implementation specific file to another implementation. These functions are to be created the replicated service itself, because it is only the service that has initial access to the files generated by the replicas. Then, after converting the file to fit the implementation of the replica that is being recovered, the replicated service invokes the import function offered by the implementation, passing the already converted file.

The process of re-populating the recovering replica's database ends with the execution of the sql commands of the exported file.

Importing the filter

The last step in recovering a faulty-replica is the importation of the filter. This process takes place at the recovering replica. Although, a serialized version of the filter has successfully been created at a correct replica, it still needs to be passed to the recovering replica, in order for this replica to re-create its filter. Given that all interactions with the replicas are made using the *JDBC* interface, transferring the serialized version of the filter between a correct replica and the one being recovered has also to be achieved using this interface.

The idea behind the process of importing the serialized version of the filter is similar to the process used in the replicated service for identifying faulty-replicas. Where a specific table is accessed and the *token* that represents the replica's state is retrieved by the replicated service. The same is done when importing a filter. Where the replicated services accesses table T_f of a correct state replica, and it is the service's responsibility to pass this value to the recovering replica. Passing this serialized version of the filter to the recovering replica happens by using the *JDBC* interface, the filter is passed to the replica through its table T_f.

After the filter has been serialized and exported at a correct replica, it is inserted in the recovering replica's table T_f. Inserting this serialized version of the filter in table T_f raises another problem. Due to the fact that this insert operation is no different from other insert operations, if it was to be treated in the same way by the recovering replica, this would have this replica buffering this transaction, and going through the normal routine of later using the values from this transaction in its filter, hence this insert operation has to be filtered. This is done by modifying the behavior of both databases, treating this transaction as a special case. When a transaction arrives at the database, it is checked to see if the affected table is T_f, and if so, it is to be treated differently. Upon identifying that the operation of that transaction is an insert operation in table T_f, a process for recreating the structure responsible for creating the replica's state summarization *token* is triggered.

The process of recreating this structure is similar to the one that occurs when creating a serialized version of the filter at the correct replica. The value that was inserted in table T_f is used to recreate the state summarization structure — the filter. This value is a string of text that corresponds to the serialized version of a correct filter, having associated with each table a Scalable Counting Bloom Filter. Recreating the structure is done by parsing this value. Each of the table's in the string is accessed and for each of them, it is invoked the function that recreated the Scalable Counting Bloom Filter, passing to this function a serialized version of the Scalable Counting Bloom Filter of that table. This function then recreates the table's Scalable Counting Bloom Filter by creating a new Scalable Counting Bloom Filter, using as initialization parameters the variables' values that were serialized.

Additionally, given that a Scalable Counting Bloom Filter entails the presence of sub-filters, they too need to be recreated. This process is identical to the aforementioned one. Where the Counting Bloom Filters — sub-filters — are initialized using the values present in string of text. One small difference between these two processes, is that the Counting Bloom Filter has an array of bytes, which has itself been serialized. Thus, in order to complete the recreation of the sub-filters, the serialized byte arrays also need to be recreated, this is done by iterating the serialized version, and initialing a new byte array with the corresponding correct values.

After all tables have recreated their Scalable Counting Bloom Filter, the recovering replica has now completed the recreation of the state summarization structure, and the replica has been successfully recovered.

6.2 Evaluation

This section's purpose is to analyze the implemented solution for both identifying a failed replica, and for recovering a faulty-replica. Identifying the impact that the state summarization has on a database is done by comparing the time it takes for a database to execute a certain amount of operations with the time that a modified version of the database takes to perform the same operations while also generating a state summarization *token*. The recovery of a replica is analyzed in terms of the time it takes for a correct replica to export is filter, and the database contents to a file, and the time that it takes for the recovering replica to recreate its filter, and to re-populate its database's contents using the aforementioned file.

This section is divided in three parts, the first details the methodology applied to gather the relevant information to perform the tests described in the previous paragraph, the second part is dedicated to described the obtained results, and lastly, the third part, is dedicated to the analysis of the impact that adding support for state summarization and state recovery has on both *H2 Database* and *HSQL Database*.

6.2.1 Methodology

This section describes the methodology applied to gather the data that is later used to perform an impact analysis of the implemented solution.

All of the tests described in this section were run multiple times, ensuring that the results reflected the ordinary impact that solution has in the databases. These tests were run in a Intel® Core^{TM} i7-2630QM CPU @ 2.00 GHZ, with 4GB of RAM, running Microsoft Windows 7 Professional, Java version 1.7, *H2 Database* version 1.3.170, and *HSQL Database* version 2.2.9. Additionally, the initial size for the Scalable Counting Bloom Filters used in the tests had support for 1000 objects.

The objective for the tests was to collect data that would allow to determine: i) the correctness of the state summarization solution; ii) the overhead added by the state summarization process; and iii) the overhead of the recovery procedure.

Test #1 - State Summarization

This was tested by executing the same exact n (where $n \in \{1000, 5000, 10000, 50000, 100000\}$) operations at both modified versions of the *H2 Database* and *HSQL database*. By having each database execute these variable amounts of operations, we will be able to retrieve data that allows for the correctness of the state summarization solution to be attained.

Although the Scalable Counting Bloom Filter supports three kinds of operations: insert, delete, and update; the test only encompassed operations of the first type: insert.

Test #2 - State Summarization Overhead

This test was executed in the original versions of the *H2 database* and *HSQL database*, but also in the modified versions — those with support for state summarization. This allows for a direct comparison between the time it takes for the un-modified versions run the tests and the modified versions, thus allowing to understand the overhead imposed by the state summarization process.

In this test the same methodology used for the previous test was applied. At each database the same n (where $n \in \{1000, 5000, 10000, 50000, 100000\}$) operations were executed.

Test #3 - State Recovery

This test's goal is to gather data that allows for the understanding if the state recovery process implemented is correct. This is done by having executing n (where $n \in \{1000, 5000, 10000, 50000, 100000\}$) operations in the modified versions of the databases, and verifying that the state summarization *token* that is generated after the recovery of the database completes is the equal to the *token* that gets generated by running those same n operations without triggering the recovery process. Here insert operations were also used to conduct this experiment, confirming to the type of operations used in the other tests.

Due to the fact that the recovery process is divided in four stages, the data from each of this stages will also be used to perform the impact analysis. These four stages are: i) filter exported; ii) filter importation; iii) database exportation; and iv) database importation.

The first two stages allow to attain the time it takes for a correct replica to export its filter, and the time that the replica that is being recovered takes to import the filter. Additionally more operations are executed at the database, which allows for a deeper understanding of the impact that the filter size has in the time it takes for it to be imported/exported. The idea behind running more operations is that the more operations that are performed, the larger the filter size will be, thus the more sub-filters that are created in the Scalable Counting Bloom Filter. This is important to attain the impact that larger filter sizes have in the databases, as it is expected that large filters will take significantly more time to export/import.

The last two stages are meant to gather information regarding the time it takes for the databases to be exported/imported. It is expected that the time it takes for a database be to exported/imported is linear, no further operations are here performed. As our intuition is that having more operations executed at the databases, will not generate additional data that is relevant for the context of this work.

The fact that the un-modified versions of the *H2 database* and *HSQL database* do not support the concept of a filter, nulls the any possible overhead comparison between themselves and the modified versions. Which means that the data generated by this tests is to be used solely to analyze the time it takes for the modified databases to complete the recovery process.

6.2.2 Results

This section presents the data gathered from running the aforementioned tests for *H2 Database* and *HSQL Database*.

Test #1 - State Summarization

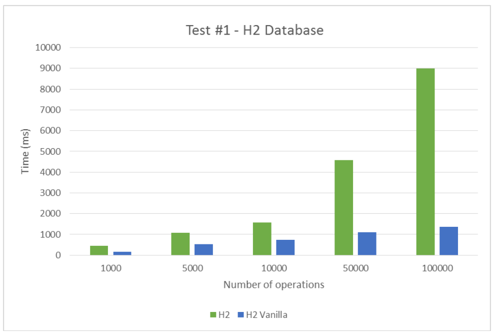
This test involved running n operations (where $n \in \{1000, 5000, 10000, 50000, 100000\}$) in both modified versions of the databases.

The implemented state summarization solution is considered correct if the state summarization *token* generated by the Scalable Counting Bloom Filter is the same for both databases for every n operations executed.

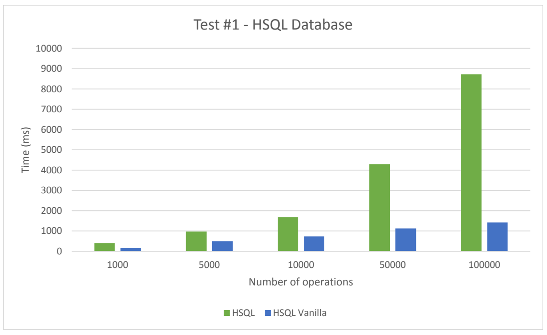
The results obtained in this test revealed that the implemented solution for state summarization works as expected. Independently of the database that executes the n operations, the state summarization *token* that is generated by using the Scalable Counting Bloom Filter is the identical. This means that the purposed solution is suitable to be used in the context of a replicated in-memory database service, as it is capable of creating an abstraction of state that can be used by the service to identify replicas that diverge during operation by comparing their state summarization *tokens*.

Test #2 - State Summarization Overhead

The charts in Figure 6.2 show average times taken to complete the test. The chart on the left compares the time taken by the vanilla *H2 Database* to complete the test with the time taken by the modified *H2 Database*. As expected, given that extra operations — related to the state summarization *token* generation — are executed at the modified version, it also takes more time for the database to execute the same operations as the vanilla version. We see that despite the number of operations the vanilla *H2 Database*'s time does not follow a linear curve, whereas the modified version does, this is justified by the fact that the inherent complexity added by the Scalable Counting Bloom Filter to the modified version, where it has to generate hashes when executing the operations, thus taking more time to complete the execution of the test.



(a) Test #1 - H2 Database results



(b) Test #1 - HSQL Database results

Figure 6.2: Test #2 - State Summarization Overhead Results

The reason behind the modified version taking so much more time to complete the test, particularly in the case for $n \in \{50000, 10000\}$, is justified by the number of times that the Scalable Counting Bloom Filter is expanded. The more filter expansions that take place, the higher the overhead imposed by the state summarization process. The bigger the Scalable Counting Bloom Filter gets, the larger the sub-filters created, affecting the time it will take for further database operations to be reflected by the filter. This is due to the fact that for large sub-filters — Counting Bloom Filters — the number of hashes that need to be generated to achieve the representation of the object that is to be added is higher than when compared to a Counting Bloom Filter of smaller size. By having to generate more hashes to calculate the positions in the sub-filter that will be used to store the representation of the element, the process will take longer to complete. Additionally, by having more sub-filters, the membership test that takes place when adding a new element will also take longer, as there are more sub-filters that need to be tests for membership.

This is independent from the type of operation being performed by the database, because when the time comes for block of buffered transactions to be reflected by the filter, for each object that is to used in the filter, the same amount of hashes are generated, whether it be an insert, delete, or update.

The same results were obtained when the test was executed at the *HSQL Database*, as shown by the chart on the right side of Figure 6.2. Here we see that for the modified *HSQL Database*, the time taken for execute n operations (where $n \in \{1000, 5000, 10000, 50000, 100000\}$) follows a linear curve, whereas the vanilla *HSQL Database* does not.

The reason for this is the same that was given for the *H2 Database*. It is related to the Scalable Counting Bloom Filter itself, and the amount of sub-filters that it contains. The more operations that are executed, the more operations need to be reflected by the filter, which eventually translates into the filter having to expanded at some point in time. During this test, the filter will reach moments where it needs to expand, making room for further objects. The more operations that are performed, the more times the filter will reach moments where it needs to expand. This means that for larger filter sizes

— which implies that there are more sub-filters — the number of hashes that needed to be generated to reflect a database operation in the filter will be larger. Not only for the addition of the operation itself, but also for the membership test, which is what allows the filter to correctly identify in which sub-filter is the element to be added.

Test #3 - State Recovery

The results from this test are segmented in four parts: i) filter exportation; ii) filter importation; iii) database exportation; and iv) database importation;

Figure 6.3 shows the results for the first part of the test — filter exportation. As previously stated, the filter exportation process is the process of creating a serialized version of the various Scalable Counting Bloom Filters present in the database. In the context of this text this means that the database has to access its only table — the one used to execute the operation — and export that filter. The process of exporting a Scalable Counting Bloom Filter is essentially done by exporting its sub-filters and its variables, which means that it is expected that the more sub-filters that need to be exported — converted to a string representation — the more that it will take for the database to complete the filter exportation process.

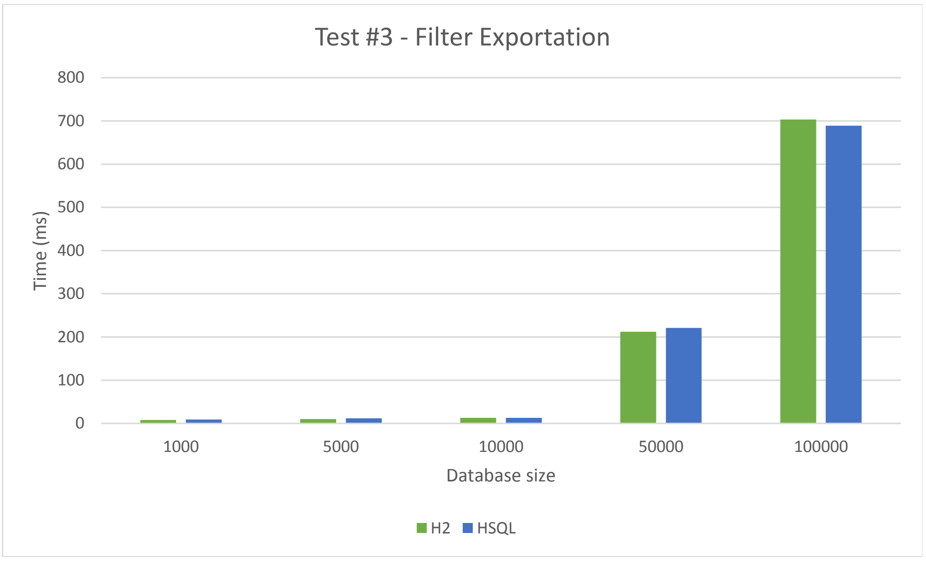


Figure 6.3: Test #3 - Filter Exportation Results

The chart in the figure reflects exactly this. For $n = 1000$ it takes little to no time to perform the filter exportation, as here only one sub-filter exists in the Scalable Counting Bloom Filter. As the value n grows, the number of expansions that occur at the Scalable Counting Bloom Filter increase, and for each expansion that occurs a new sub-filter is created. And, as expected, the time that is required to export the filter in this scenarios is much higher. In fact, the time difference between the exportation of a filter of size $n = 10000$ and a filter of size $n = 50000$ is orders of magnitude higher, the same remaining true for the filter of size $n = 100000$. These results, for $n = 50000$ and $n = 100000$, seem strange, and we could not find a justification for this happening, as two filters a and b, where b's size is twice as large as the size of a should take at most double the time to be exported, which clearly is not happening for the two larger datasets used in this test.

This chart also shows that the process is independent of the databases. Each of the databases used take approximately the same time to complete the filter exportation process. This fact was also predictable, as not only is the state summarization structure the same for both databases, but also the process of exporting this structure is the same. Here, implementation specific aspects of the databases are irrelevant, as the portion of code that executes the exportation of the filter, is exactly the same.

The char in Figure 6.4 shows the results for the second part of the test: filter importation. In this part of the test, each database used the filter that was exported in the first step, to recreate its state summarization structure — Scalable Counting Bloom Filter. Recreating this structure is done by parsing the serialized filter, and extracting the various pairs $< T, F >$. Where T is the name of a table, and F is its serialized Scalable Counting Bloom Filter, in the form depicted in Figure 6.1. For update operations, where we need to first remove the old object's representation that was modified from the filter, and then insert the new modified value, we will have two pairs for this type operation. Given that both databases execute updates by removing the old row and then inserting the modified row, the result from updating a row can be reflected in the Scalable Counting Bloom Filter as we will have two pairs $< T, F >$ that allow the filter to first remove the old object representation and then insert the new one.

In this test only one table was used, therefore there is only one pair $< T, F >$. After extracting this pair, the database accesses table T, and invokes the method that imports the filter F. This method parses this serialized Scalable Counting Bloom Filter F, and initializes a new Scalable Counting Bloom Filter in table T with the parameters that were contained in F.

Additionally, this process is also going to re-create the structures of the Scalable Counting Bloom Filter. Given that a Scalable Counting Bloom Filter can be seen as a bucket for Counting Bloom Filters, the filter F also contains these sub-filters in a serialized form. This requires that the method responsible for re-creating the Scalable Counting Bloom Filter at table T to also re-create the sub-filters. This is done by extracting from filter F the sub-filters associated with the filter that is being recreated. Similarly to the aforementioned process, here the sub-filters are extracted from filter F. The value from the extraction process can be seen as a pair $< V, B >$, where V corresponds to list of all the variables needed to initialize the Counting Bloom Filter, and B is the serialized version of the sub-filter's byte array

This step terminates when the Scalable Counting Bloom filter has been re-created.

What the results from this step show is that the larger the filter that is being imported is, the longer it takes for the database to complete the step. Also, the results show that the time that is needed for the database to import a filter of size n (where $n \in \{1000, 5000, 10000, 50000, 100000\}$) — here a filter of size n is a filter that has executed

n operations — is approximately the same time that is needed to perform the first step of the test. The reason for this happening is the fact the task of converting a sub-filter's serialized byte array back to the original byte array is the same time taken for the inverse operation.

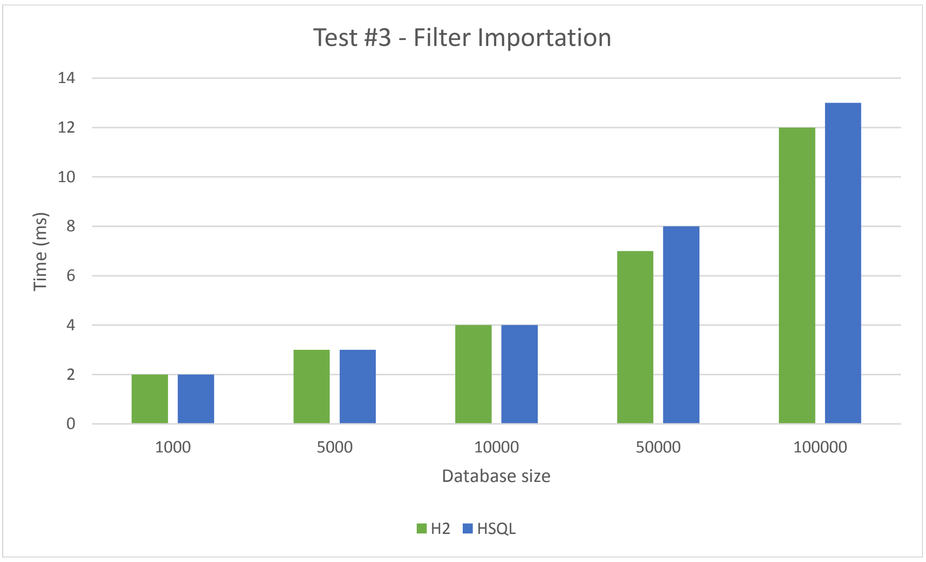


Figure 6.4: Test #3 - Filter Importation Results

Figure 6.3 shows the time taken by the modified versions of *H2 Database* and *HSQL Database* to perform the filter exportation for different database's sizes. Both databases appear to execute the filter exportation in approximately the same amount of time. The reason behind this is that the process for exporting the filter is the same for both databases, not being directly influenced by any implementation specific protocol. In this chart it can also be seen that as the filter grows — the more operations that were performed — the amount of time it takes to export the filter also grows, which was to be expected. This is due to the fact that for larger filters they have more sub-filters, and by having not only more sub-filters, but also increasingly bigger sub-filters, the time required for exporting said filters will be higher.

Figure 6.5 shows the average times for the database exportation process, for both *H2 Database* and *HSQL Database*. The results show that there is a clear difference between the time it takes *H2 Database* to export its database's contents to a file and the time required by *HSQL Database* to complete the process for the same values of n (where $n \in \{1000, 5000, 10000, 50000, 10000\}$).

This graph also confirms that the larger the database — which entails the presence of larger Scalable Counting Bloom Filters — the more time is required to export the database, as there is more content to be exported. Although both databases show to have very different performances, the results show that the time taken to perform the exportation of the database follows a linear curve.

These results allow also for the understanding that if a replica fails, and needs to

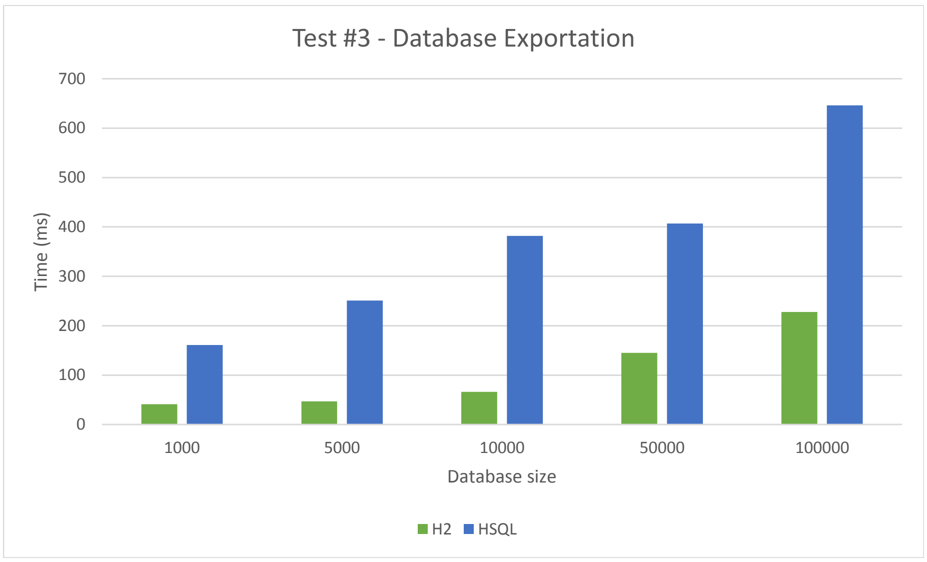


Figure 6.5: Test #3 - Database Exportation Results

undergo the recovering procedure, it is best for the replicated service to request database to be exported by a replica running an *H2 Database* (if available). Which will make the recovery of that replica faster, as the replica will have to wait less time for a database to be exported.

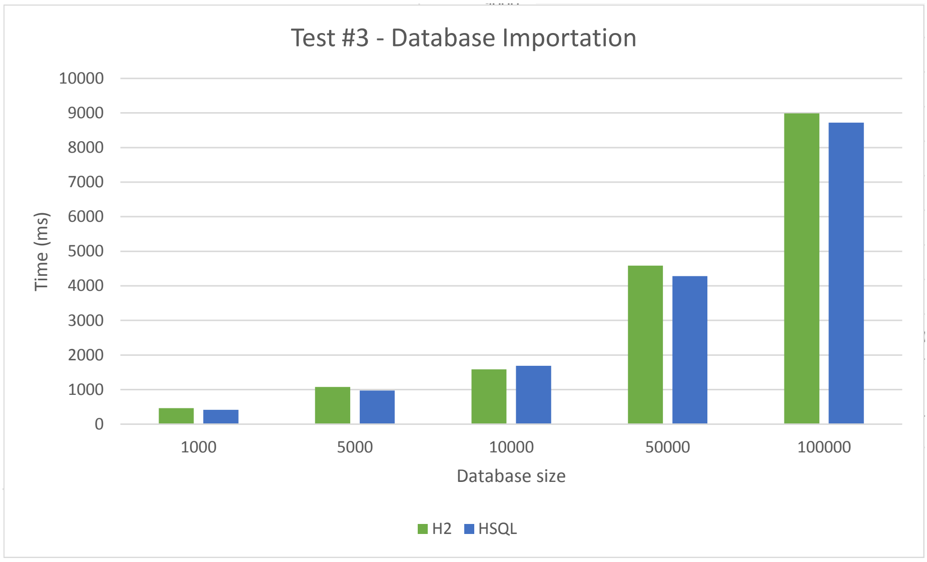


Figure 6.6: Test #3 - Database Importation Results

Finally, the results from the last step of the recovery process are shown in Figure 6.6. This chart shows the average time taken by both modified versions of *H2 Database* and *HSQL Database* to import the database contents previously exported to a file. We see that independently of the number of operations — database size — that the file contains the *H2 Database* requires less time to execute the operation. The reason for this might be associated with the fact that the *HSQL Database* does not offer any direct method to execute a *sql script*, despite offering the opposite method, and the file had to be manually parsed in order to successfully execute each sql command present in the file.

Another aspect to note is that the time taken to complete this task, for each value of n (where $n \in \{1000, 5000, 10000, 50000, 100000\}$), is similar to the time obtained in Test #1. This is due to the fact that during this process the database will also be computing the state summarization *token*, as these operations do not differ from any other operations, thus the database is not able to distinguish that in this case the state summarization process is not to take place. By computing the state summarization *token* during a database importation, the database is effectively wasting resources, as the *token* generated will be replaced when the filter is imported. In the final chapter, a possible improvement to this process is suggested.

7 Conclusion

This work presented a solution for summarizing the state of a database, and for performing state recovering in case of failure, that can be used in the context of replicated services, allowing for a greater resilience to failures. The state summarization is achieved by using Scalable Counting Bloom Filters, which provide a way for summarizing the state of a replica, and by also using Merkle Trees, that provide an efficient way for summarizing the filter itself.

By providing the mechanisms for the state of a database to be summarized, the replicated service can identify replicas that diverge during operation by comparing the state summarization *tokens* of the replicas. Those replicas whose *tokens* do not match the majority of the *tokens* can be marked by the replicated service as having diverged from the others, thus having become faulty, and needing to undergo the recovery process (state recovery). Our solution addresses a problem with current summarization techniques, such as Merkle Trees — they work well only if the service maintains the exact same internal state for the same data. In many situation this might not be the case — e.g. in Red-Black trees and other dynamic structures, the internal representation will vary depending on the order by which operations were executed. The same problem occurs if different implementations are used in different replicas. To address this problem, the typical solution is to compute the summary of the state based on an abstract state. However, this is rather complex or inefficient.

Our solution addresses this issue, as the summary depends directly on the service state. Although our solution also does not allow data to be added in a completely different order, we propose a mechanism for allowing sets of operations to execute concurrently without impacting the produced summary.

The recovery process is done in stages: i) exporting the filter of a correct replica; ii) dumping the contents of the database; iii) importing said data dump to the replica that is being recovered; and, finally, iv) recreating the filter of that replica by importing the previously exported filter. Effectively recovering the state of a faulty-replica, as both its contents and its filter are identical to the ones present at a correct replica, after the recovery process completes.

We have implemented our solution in two in-memory databases — *H2 Database* and *HSQL Database*. We described the technical challenges we faced, and the rationale for the decision taken. We have evaluated our solution experimentally — the reported results show that the proposed approach correctly summarizes the state of the database in both replicas, reaching the same summary. Our approach has a non-negligible overhead that must be tackled before it can be used in practice.

7.1 Future Work

One characteristic that the implemented solution lacks is its inability to adapt the initial Scalable Counting Bloom Filter size of the tables, which affects the performance of the structure. It would be interesting to implement a mechanism to define for any given table its initial filter size, as the current solution can lead to unnecessary filter expansions due to inappropriate filter sizes, which wastes system resources as all tables — big or small — will be forced to use the same filter size. By implementing the aforementioned mechanism tables that were expected to be small could be configured to have smaller filter sizes, and the filters of bigger tables configured to have bigger sizes, thus restricting the occurrence of wasteful filter expansions.

Additionally, the recovering mechanism can still be further improved. The implemented solution first imports the contents of a correct state replica's database, and then it imports the filter of that replica. The problem here is that the replica does not differ the insert operations in this process from other routine insert operations, which means that it will go through the normal process of buffering transactions and computing the state summarization *token*, effectively wasting resources, as the *token* generated here will be replaced when the filter importation takes place. One possible solution to improve this process is to stop the state summarization process during this process.

Another aspect of the recovery mechanism that can be improved is the filter exportation process. When a sub-filter is exported, its serialized representation can be cached, allowing future exportations to be faster, as this buffered representation will still be valid until an operation is performed at its corresponding sub-filter.

Finally, the performance impact of the solution can also be improved. One possible solution to address the non-negligible impact that the current prototype has is to possible exploit the capabilities of multi-core machines and use a background thread to update the filter, while the main thread remains undisturbed.

Bibliography

[1] G. Moore. *Excerpts from a conversation with Gordon Moore: Moore's Law*. [Accessed on Jan. 31, 2013]. 2005.

[2] G. E. Moore. "Readings in computer architecture". In: ed. by M. D. Hill, N. P. Jouppi, and G. S. Sohi. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. Chap. Cramming more components onto integrated circuits, pp. 56–59.

[3] D. Geer. "Industry Trends: Chip Makers Turn to Multicore Processors". In: *Computer* 38.5 (May 2005), pp. 11–13.

[4] J. Soares, P. Mariano, J. Loureço, and N. Preguiça. "Improving Application Performance with Diverse Component Replication". In: Submitted for publication.

[5] J. a. Soares, J. a. Lourenço, and N. Preguiça. "MacroDB: scaling database engines on multicores". In: *Proceedings of the 19th international conference on Parallel Processing*. Euro-Par'13. Aachen, Germany: Springer-Verlag, 2013, pp. 607–619.

[6] F. B. Schneider. "Implementing fault-tolerant services using the state machine approach: a tutorial". In: *ACM Comput. Surv.* 22.4 (Dec. 1990), pp. 299–319.

[7] P. J. Marandi, M. Primi, and F. Pedone. "High performance state-machine replication". In: *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*. DSN '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 454–465.

[8] M. Castro, R. Rodrigues, and B. Liskov. "BASE: Using abstraction to improve fault tolerance". In: *ACM Trans. Comput. Syst.* 21.3 (Aug. 2003), pp. 236–269.

[9] A. Avizienis. "The N-Version Approach to Fault-Tolerant Software". In: *IEEE Trans. Softw. Eng.* 11.12 (Dec. 1985), pp. 1491–1501.

[10] A. Avizienis, J.-C. Laprie, B. Randell, and Vytautas. "Fundamental Concepts of Dependability". In: (2000).

[11] F. B. Schneider. "Byzantine generals in action: implementing fail-stop processors". In: *ACM Trans. Comput. Syst.* 2.2 (May 1984), pp. 145–154.

[12] M. Castro and B. Liskov. “Proactive recovery in a Byzantine-fault-tolerant system”. In: *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*. OSDI’00. San Diego, California: USENIX Association, 2000, pp. 19–19.

[13] N. Kolettis and N. D. Fulton. “Software Rejuvenation: Analysis, Module and Applications”. In: *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*. FTCS ’95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 381–.

[14] J. P. J. Kelly, T. I. McVittie, and W. I. Yamamoto. “Implementing Design Diversity to Achieve Fault Tolerance”. In: *IEEE Softw.* 8.4 (July 1991), pp. 61–71.

[15] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. “Tolerating latency in replicated state machines through client speculation”. In: *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. NSDI’09. Boston, Massachusetts: USENIX Association, 2009, pp. 245–260.

[16] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. “Database engines on multicores, why parallelize when you can distribute?” In: *Proceedings of the sixth conference on Computer systems*. EuroSys ’11. Salzburg, Austria: ACM, 2011, pp. 17–30.

[17] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. “All about Eve: execute-verify replication for multi-core servers”. In: *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI’12. Hollywood, CA, USA: USENIX Association, 2012, pp. 237–250.

[18] J. Soares, J. Lourenço, and N. M. Preguiça. “Software Component Replication for Improved Fault-Tolerance: Can Multicore Processors Make It Work?” In: *Proceedings of the 14th European Workshop on Dependable Computing (EWDC), LNCS 7869*. 2013, pp. 173–180.

[19] C. Plattner and G. Alonso. “Ganymed: scalable replication for transactional web applications”. In: *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Middleware ’04. Toronto, Canada: Springer-Verlag New York, Inc., 2004, pp. 155–174.

[20] K. Daudjee and K. Salem. “Lazy database replication with snapshot isolation”. In: *Proceedings of the 32nd international conference on Very large data bases*. VLDB ’06. Seoul, Korea: VLDB Endowment, 2006, pp. 715–726.

[21] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. “Detecting and surviving data races using complementary schedules”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. Cascais, Portugal: ACM, 2011, pp. 369–384.

[22] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn. “Respec: efficient online multiprocessor replayvia speculation and external determinism”. In: *SIGPLAN Not.* 45.3 (Mar. 2010), pp. 77–90.

[23] M. Castro and B. Liskov. "Practical Byzantine fault tolerance". In: *Proceedings of the third symposium on Operating systems design and implementation*. OSDI '99. New Orleans, Louisiana, USA: USENIX Association, 1999, pp. 173–186.

[24] I. Gashi, P. Popov, and L. Strigini. "Fault Tolerance via Diversity for Off-the-Shelf Products: A Study with SQL Database Servers". In: *IEEE Trans. Dependable Secur. Comput.* 4.4 (Oct. 2007), pp. 280–294.

[25] I. Gashi, P. Popov, V. Stankovic, and L. Strigini. "On Designing Dependable Services with Diverse Off-the-Shelf SQL Servers". In: Lecture Notes in Computer Science 3069 (2004). Ed. by R. Lemos, C. Gacek, and A. Romanovsky, pp. 191–214.

[26] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. "An Improved Construction for Counting Bloom Filters". In: *Algorithms – ESA 2006*. Ed. by Y. Azar and T. Erlebach. Vol. 4168. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 684–695.

[27] A. Broder and M. Mitzenmacher. "Network applications of bloom filters: A survey". In: *Internet Mathematics* 1.4 (2004), pp. 485–509.

[28] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. "Scalable Bloom Filters". In: *Inf. Process. Lett.* 101.6 (Mar. 2007), pp. 255–261.