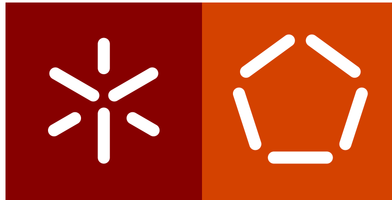


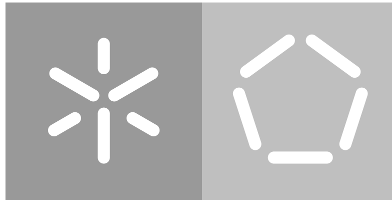
Universidade do Minho
Escola de Engenharia
Departamento de Informática

Hugo Miguel Ferreira Abreu

High Availability Architecture for Cloud Based Databases

Arquitetura de Elevada Disponibilidade para Bases de Dados na Cloud

November 2019



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Hugo Miguel Ferreira Abreu

High Availability Architecture for Cloud Based Databases

Arquitetura de Elevada Disponibilidade para Bases de Dados na Cloud

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Professor Doutor José Orlando Roque Nascimento Pereira
Doutor Fábio André Castanheira Luís Coelho

November 2019

Despacho RT - 31 /2019 - Anexo 3

Declaração a incluir na Tese de Doutoramento (ou equivalente) ou no trabalho de Mestrado

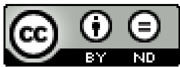
DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Atribuição-SemDerivações
CC BY-ND

https://creativecommons.org/licenses/by-nd/4.0/

ACKNOWLEDGEMENTS

It has been a long path since the start of this journey, and yet another chapter is closed. This dissertation is not only possible by my efforts. First of all, I would like to thank all my advisors, Prof. José Orlando Pereira whose help and advice were of the greatest importance towards the final work and results. Without his guidance, this wouldn't be possible. The commitment, encouragement, and guidance given by my advisor Prof. Fábio André Coelho were meaningful and extraordinary, providing is uppermost important supervision every day. Likewise, to Prof. Ana Nunes Alonso whose critical opinions allowed us to move forward and achieve the goals desired, by helping in every decision. I would also like to thank for the opportunity to work with this great team, allowing me to grow not only academically, professionally, but also as a person.

I would also like to thank my working colleagues who had always been there to help and give friendly advices. Our understanding and hard work led us to this place. I must thank you all for your support and sympathy.

I would also like to thank my closest friends. They have always been there, helping me to distance from the problems when needed, refreshing the ideas and giving new visions, considering life is not exclusively work.

Finally, I would like to dedicate this work, and life achievement to my parents who always supported me throughout all my academic and, personal life. They strived to help me follow my passion and, encouraged me to succeed.

Thank you.

This work was partially funded by FCT - Fundação para a Ciência e a Tecnologia, I.P., (Portuguese Foundation for Science and Technology) within project UID/EEA/50014/2019

Despacho RT - 31 /2019 - Anexo 4

Declaração a incluir na Tese de Doutoramento (ou equivalente) ou no trabalho de Mestrado

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

RESUMO

Com a constante expansão de sistemas informáticos nas diferentes áreas de aplicação, a quantidade de dados que exigem persistência aumenta exponencialmente. Assim, por forma a tolerar faltas e garantir a disponibilidade de dados, devem ser implementadas técnicas de replicação.

Atualmente existem várias abordagens e protocolos, tendo diferentes tipos de aplicações em vista. Existem duas grandes vertentes de protocolos de replicação, protocolos genéricos, para qualquer serviço, e protocolos específicos destinados a bases de dados. No que toca a protocolos de replicação genéricos, as principais técnicas existentes, apesar de completamente desenvolvidas e em utilização, têm algumas limitações, nomeadamente: problemas de performance relativamente a saturação da réplica primária na replicação passiva e o determinismo necessário associado à replicação ativa. Algumas destas desvantagens são mitigadas pelos protocolos específicos de base de dados (e.g., com recurso a multi-master) mas estes protocolos não permitem efetuar uma separação entre a lógica da replicação e os respetivos dados. Abordagens mais recentes tendem a basear-se em técnicas de replicação com fundamentos em mecanismos distribuídos de *logging*. Tais mecanismos proporcionam alta disponibilidade de dados e tolerância a faltas, permitindo abordagens inovadoras baseadas puramente em *logs*.

Por forma a atenuar as limitações encontradas não só no mecanismo de replicação ativa e passiva, mas também nas suas derivações, esta dissertação apresenta uma solução de replicação híbrida baseada em middleware, o SQLware. A grande vantagem desta abordagem baseia-se na divisão entre a camada de replicação e a camada de dados, utilizando um log distribuído altamente escalável que oferece tolerância a faltas e alta disponibilidade. O protótipo desenvolvido foi validado com recurso à execução de testes de desempenho, sendo avaliado em duas infraestruturas diferentes, nomeadamente, um servidor privado de média gama e um grupo de servidores de computação de alto desempenho. Durante a avaliação do protótipo, o standard da indústria TPC-C, tipicamente utilizado para avaliar sistemas de base de dados transacionais, foi utilizado. Os resultados obtidos demonstram que o SQLware oferece uma aumento de throughput de 150 vezes, comparativamente ao mecanismo de replicação nativo da base de dados considerada, o PostgreSQL.

Palavras-chave: middleware, replicação, logs, logs distribuídos, bases de dados, replicação híbrida, replicação ativa, rpelicação passiva, tolerância a faltas, alta disponibilidade.

ABSTRACT

With the constant expansion of computational systems, the amount of data that requires durability increases exponentially. All data persistence must be replicated in order to provide high-availability and fault tolerance according to the surrogate application or use-case.

Currently, there are numerous approaches and replication protocols developed supporting different use-cases. There are two prominent variations of replication protocols, generic protocols, and database specific ones. The two main techniques associated with generic replication protocols are the active and passive replication. Although generic replication techniques are fully matured and widely used, there are inherent problems associated with those protocols, namely: performance issues of the primary replica of passive replication and the determinism required by the active replication. Some of those disadvantages are mitigated by specific database replication protocols (e.g., using multi-master) but, those protocols do not allow a separation between logic and data and they can not be decoupled from the database engine. Moreover, recent strategies consider highly-scalable and fault-tolerant distributed logging mechanisms, allowing for newer designs based purely on logs to power replication.

To mitigate the shortcomings found in both active and passive replication mechanisms, but also in partial variations of these methods, this dissertation presents a hybrid replication middleware, SQLware. The cornerstone of the approach lies in the decoupling between the logical replication layer and the data store, together with the use of a highly scalable distributed log that provides fault-tolerance and high-availability. We validated the prototype by conducting a benchmarking campaign to evaluate the overall system's performance under two distinct infrastructures, namely a private medium class server, and a private high performance computing cluster. Across the evaluation campaign, we considered the TPC-C benchmark, a widely used benchmark in the evaluation of *Online transaction processing (OLTP)* database systems. Results show that SQLware was able to achieve 150 times more throughput when compared with the native replication mechanism of the underlying data store considered as baseline, PostgreSQL.

Keywords: middleware, replication, logs, distributed logs, databases, hybrid replication, active replication, passive replication, fault tolerance, high availability.

CONTENTS

1 INTRODUCTION 1
- 1.1 Problem 3
- 1.2 Objectives 4
- 1.3 Thesis structure 4

2 BACKGROUND 5
- 2.1 Replication techniques 5
  - 2.1.1 Active Replication 5
  - 2.1.2 Passive Replication 6
  - 2.1.3 Semi-Active Replication 8
  - 2.1.4 Semi-Passive Replication 9
  - 2.1.5 Hybrid Transactional Replication 11
- 2.2 Database specific replication protocols 12
  - 2.2.1 Conservative execution 12
  - 2.2.2 Optimistic execution 14
  - 2.2.3 Hybrid execution 15

3 MIDDLEWARE-BASED REPLICATION MECHANISM 17
- 3.1 Database Replication Systems 17
  - 3.1.1 PostgreSQL Logical Replication 17
  - 3.1.2 Amazon Aurora 20
- 3.2 Distributed Logs 21
  - 3.2.1 Architecture 22
  - 3.2.2 Log Writers 23
  - 3.2.3 Log Readers 24
  - 3.2.4 Log Streams Replication 25
- 3.3 Discussion 26

4 SQLWARE REPLICATION MIDDLEWARE 29
- 4.1 Middleware 29
- 4.2 Specification 31
  - 4.2.1 Backlog 33
  - 4.2.2 Assumptions 35
  - 4.2.3 Horizontal partitioning 35
  - 4.2.4 Underlying Database Management System 36
  - 4.2.5 Configuration parameters 36

5 SYSTEM ANALYSIS AND RESULTS 39
5.1 Experimental setting 39
5.2 Configurations 41
5.3 Results 43
5.3.1 Single Replica environment 43
5.3.2 High Performance Computing environment 46
5.4 Discussion 48
6 CONCLUSIONS AND FUTURE WORK 50
6.1 Conclusion 50
6.2 Future Work 51

LIST OF FIGURES

Figure 1 Active replication mechanism. 6
Figure 2 Passive replication mechanism. 7
Figure 3 Semi-Active replication mechanism. 8
Figure 4 Semi-Passive replication mechanism. 10
Figure 5 Hybrid transactional replication mechanism. 12
Figure 6 Database conservative replication. 13
Figure 7 Database optimistic PostgresR replication. 14
Figure 8 Database optimistic DBSM replication. 15
Figure 9 PostgreSQL logical replication slots. 18
Figure 10 PostgreSQL *synchronous_commit* configuration values. 19
Figure 11 Last Add Confirmed Mechanism. 22
Figure 12 Apache BookKeeper log segment allocation. 25
Figure 13 Protocols comparison within a quadrant. 26
Figure 14 Hybrid middleware based replication: Comprehensive view. 30
Figure 15 Hybrid middleware architecture flow. 31
Figure 16 SQLware active replication. 32
Figure 17 SQLware passive replication. 33
Figure 18 Backlog critical architectural points. 34
Figure 19 Single machine benchmarks comparison. 45
Figure 20 *High Performance Computing (HPC)* benchmarks comparison. 47

LIST OF TABLES

Table 1 SQLware underlying database configuration parameters. 37
Table 2 SQLware underlying database write-set service parameters 37
Table 3 SQLware DistributedLog API configuration. 38
Table 4 SQLware configuration values for the single machine setup. 42
Table 5 SQLware configuration values for the HPC setup. 43
Table 6 PostgreSQL synchronous replication benchmark values. 43
Table 7 PostgreSQL asynchronous replication benchmark values. 44
Table 8 Middleware replication benchmark values. 44
Table 9 PostgreSQL synchronous replication benchmark values on HPC. 46
Table 10 Hybrid replication benchmark values on HPC. 47

ACRONYMS

***DIV*consensus** *Consensus with Deferred Initial Values*.

ABCAST Atomic Broadcast.
API Application Programming Interface.

DBSM Database State Machine.
DDL Data Definition Language.
DLSN DistributedLog Sequence Number.
DU Deferred-Update.

HPC High Performance Computing.
HTR Hybrid transactional replication.

IaaS Infrastructure as a Service.

JDBC Java Database Connectivity.

LAC LastAddConfirmed.
LAP LastAddPushed.
LSN Log Sequence Number.

ODBC Open Database Connectivity.
OLTP Online transaction processing.

Pub-Sub Publisher-Subscriber.

SaaS Software as a Service.
SM State-Machine.

URI Uniform Resource Identifier.

V-JDBC Virtual Java Database Connectivity.
VCL Volume Complete LSN.
VSCAST View synchronous multicast.

WAL Write-Ahead Log.
WH Warehouse.

1

INTRODUCTION

The worldwide expansion of the internet lead to a large growth of web clusters and cloud services that offer *Infrastructure as a Service (IaaS)*. Deploying databases in an IaaS environment is a reasonable trade-off, enabling companies to focus on their core business instead of expending resources on computer infrastructure and maintenance, taking advantage of the *pay-as-you-go* model.

Statistics collected by Wordpress show that each month there are 74 million new posts and around 57 million new comments in this platform [27]. Also, reddit users, in November 2018 produced 117 million comments and 14 million submissions [21]. All this data and information needs to be available at all times to guarantee a good service. Valuable information like bank transactions also requires persistence and high availability. Statistical data gathered by the European Central Bank, states that around 57 billion transactions were processed by retail payment systems [13]. Non-cash payments increased 7.9% compared with the previous year. Electronic payments require processing, verification, and persistence based on client data, and if that data becomes inconsistent or unavailable, losses can be extensive.

From a simple social network user to an international bank, entities rely heavily on their data, thus the importance of replication when it comes to sensitive information. Although replicating data is simple, maintaining data consistently across all replicas requires correct and sophisticated protocols.

Distributed databases require replication mechanisms to guarantee persistence, dependability and availability. Those mechanisms are necessary to maintain a coherent state across all service replicas. Coherence can be achieved through inter-replica communication, either actively broadcasting transactions or by propagating already executed write-sets for backup copies to apply. This ensures that machines continuously remain in the same state even when a partial system failure occurs. In the presence of (partial) failures, mechanisms must react to guarantee that data continues accessible, mitigating possible data losses and inconveniences by having operational replicas continuing to provide the service with coherent data. Even though the main focus of replication mechanisms is to provide system dependability, those same mechanisms can provide load balancing and increase system performance.

Active replication and *Passive* replication are the main approaches in what regards replication protocols. Furthermore, derivations of these techniques are actively developed and used. All these mechanisms have a well-defined set of implementations and there is no universal solution, as both mechanisms have their advantages and disadvantages.

Active replication uses a state-machine approach, where every replica starts from the same state and all replicas execute the same sequence of operations. All replicas can receive requests in this technique yet those requests require ordering beforehand. Consequently, replicas have to agree on a total order of operations and at the end of each execution the resulting state is the same throughout the cluster. This replication protocol relies on a group communication middleware where all replicas must come to an agreement towards the total-order of the operations, and after the transaction commits, all replicas reply to the client. The Active replication mechanism lacks concurrency, hence no concurrent transactions touching common data items can execute, as conflicts could occur.

Passive replication adopts the primary-backup method. In this technique the primary replica handles the request processing. After processing a request, the primary sends the updates for backup replicas to commit, with no processing required. This means that the bottleneck is in the primary replica as it processes all requests and handles all replies to the client. Issuing all requests to a single replica can cause performance issues and primary failures become visible to clients. Moreover, the primary replica selection is the protocol's responsibility, relying on a group membership primitive for the implementation.

As databases are highly concurrent servers, usually with extreme throughput, strict Service level Agreements, and transactional semantics, specialized approaches can be implemented in what regards replication protocols. Those protocols tend to be based on primary-backup techniques, and the main approaches provide a conservative and optimistic execution, as both have benefits and limitations. These techniques use a multi-master extension of the primary-backup technique, increasing performance and allowing all replicas to receive requests. Requests handled by databases are typically bound by the abstraction of a transaction. A transaction is a unit that represents a change in the database state and, typically holds a set of operations, providing isolation, consistency and correctness. Moreover, the transaction atomicity ensures that all operations within a transaction complete entirely or have no effect whatsoever.

The two main techniques regarding database replication are the optimistic and conservative execution. Both techniques operate with different execution models, the optimistic approach generally executes a transaction as soon as it arrives. Once executed, that transaction write-set is verified and, if it proves to be conflict free with other concurrent transactions touching the same data items, the transaction can effectively commit, and the changes are applied. Furthermore, if one operation of the transaction conflicts, the entire transaction aborts. Otherwise if no conflicting transactions were executed concurrently, the current

transaction simply commits in all replicas. With coarse grain conflict classes (labels that define the objects accessed by a transaction) the optimistic approach can lead to high abort rates. Considering that coarse grain conflict classes are typically represented by the tables accessed in a transaction, the probability of conflicts is consequently higher. According to the conservative approach, before execution, the replica sends the conflict classes of a transaction to other replicas. After verification, if there are no conflicting transactions, the current transaction is processed with abort-free guarantees. If there are conflicting transactions, the current transaction is sequentially scheduled according to its conflict classes. This technique leads to a processing performance cap, executing the majority of operations sequentially when using coarse grain conflict classes, as transactions often access more than one table at a time.

Database specific replication protocols are usually deployed in database management systems, making the code hard to maintain and database dependent. Therefore, database replication protocols could be controlled by a middleware, accomplishing a separation between the logic and the replication layers. Middleware replication mechanisms are supported by generic replication approaches, making those protocols database independent, easily maintainable and allowing to implement replication with different database vendors without the need to access proprietary code.

1.1 PROBLEM

On the one hand, active replication depends heavily on a total order of the requests, relying on a group communication middleware to provide such ordering. All replicas on this approach are state-machine replicas, allowing them to receive and process requests accordingly. Considering that all replicas are state-machines, operations must be deterministic, being impractical on multi-threaded servers. On the other hand, in the passive replication approach, the order and execution of requests is determined by the primary replica alone, while other replicas act as backups, allowing non-deterministic operations. However, as request handling is the primary replica's responsibility, it may become the system bottleneck.

A hybrid replication mechanism may provide a balance between both approaches. It could allow one replica to coordinate requests, dispatching updates to backup replicas, similarly to the passive protocol. At the same time, it could allow every replica to receive requests and process them like active replication, creating distinct sets of primary-backup relationships on a per-request basis. Likewise, making such shift in the replication mechanism from a global configuration to a on-request basis would allow the overall system's performance to scale, but would induce several challenges, namely on the required assurance of inter-request determinism and a intra-request conflict free execution.

1.2 OBJECTIVES

This work aims at the development of a hybrid replication middleware that ensures strong consistency and high-availability of the underlying database management system. Moreover, it does not require the database server to have native replication, since data coherence, fault-tolerant techniques and data replication are handled by the prototype. Therefore, the middleware applies the generic concept of the replication protocols, enhanced with the key features from database specific protocols. As a database specific solution, it extends the base techniques with database requirements, allowing a straightforward integration with any relational database system, without relying on proprietary software.

This work implements a prototype, details the design and specification. Besides, this thesis evaluates extensively the developed solution, aiming to obtain the maximum throughput without compromising the availability, and ultimately trying to achieve the protocol's bottlenecks.

Machine replication protocols (e.g., active replication, passive replication, and variations) are addressed throughout this document, alongside with their benefits and limitations. Also, database specific protocols are described, covering the various approaches and implementations. Consequently, SQLware's design, specification and implementation are discussed, alongside with the technologies used in the implementation, analysis, and benchmarking.

1.3 THESIS STRUCTURE

This document structure holds 6 chapters, being the first one an introduction to this dissertation, stating the problem, motivation and objectives. In Chapter 2, background and related work will be extensively described and explained, providing a solid background to allow a flexible introduction to the solution proposed.

Chapter 3 enables a deep insight into log based replication software, it also introduces the Apache DistributedLog tool and it's architecture, ultimately leading to a rich discussion about the usage of logs in replication mechanisms.

The core solution and architecture are elaborated in Chapter 4, expressing the different stages of the development, the final system architecture and corresponding implementation. Each iteration is thoroughly documented since every decision described is significant to accomplish this dissertation objective.

The analysis of the overall performance and system bottlenecks is outlined in Chapter 5. The analysis involves different components of the system, namely, hardware, benchmarks, and configurations. Chapter 6 details the reflection and final conclusion. It discusses future work and the lessons learned with this thesis.

2

BACKGROUND

2.1 REPLICATION TECHNIQUES

Nowadays there is a comprehensive number of replication techniques developed, that are deployed as part of fully matured production systems.

General purpose replication techniques make minimal assumptions about the replicated service. Specifically, they don't assume a transactional database workload. Moreover, the various existing approaches have different limitations and performance implications.

2.1.1 *Active Replication*

The active replication technique, also known as the *state-machine* approach [23], is based on different machines executing the same operations, ensuring data consistency and equal state in all replicas. Therefore, it is possible to process transactions generating high contention.

Implementing this approach requires a client to address a server group. Therefore, the client requests need to be propagated using an *Atomic Broadcast (ABCAST)* mechanism, ordering messages and ensuring total order delivery, typically transparent to the client [8] [1] [17]. Implementing active replication state-machines requires three conditions: replica coordination, agreement and order. Replica Coordination ensures that every replica receives and process the same set of requests. The agreement condition defines that all operational replicas receive every request issued. The order condition, guarantees that all replicas in the system receive and execute requests in the same order.

Using the Atomic Broadcast primitive the client issues requests to a server group. Afterwards, replicas need to coordinate ensuring that the request executes in all machines in the same order. After executing a request, all replicas must reply to the client with the outcome of the operation. This last requisite of the mechanism can become more relaxed, requiring solely a majority of equal replies from the group, mitigating byzantine faults (not addressed in this document, the interested reader is keen to consult Schneider [23, p. 6] for more details).

Figure 1 depicts the algorithm described, with three replicas and no failures assumed, presenting the core aspects of the technique.

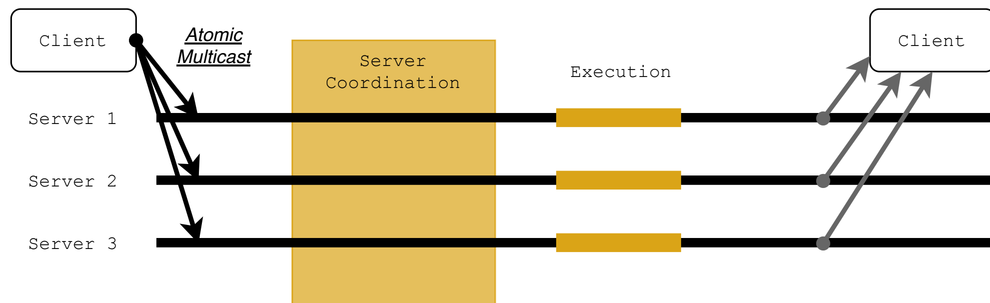


Figure 1: Active replication mechanism.

Failures are fully hidden from the client in this mechanism, as all operating replicas execute client requests and reply back, even in the presence of a (partial) failure. When a replica crashes there is no need to reissue the request.

Considering that all replicas execute the same set of transactions in the same order, deterministically, and with no concurrency, it is possible to execute irrevocable operations (e.g., system calls). Without these guarantees it would not be possible to execute such operations, since roll-back is not possible.

The biggest disadvantage of active the replication technique is the determinism required. Deterministic servers are required to boot with the same initial state and execute the same operations producing the same output. Nonetheless, multi-threading commonly results in non-determinism, leading this technique towards sequential execution. High-resource usage is another problem with this mechanism as requests are redundantly processed by all machines, with no concurrent transactions executing, bringing the processing bottleneck to one request at a time. The active replication mechanism has a well-defined set of use cases, mainly, systems handling high contention, irrevocable operations, and schemes that need to mask failures from clients (i.e., where reissues are too expensive or time-outs cannot occur). Otherwise, high consumption of resources and the determinism required by this mechanism might be counter-productive for the whole system.

2.1.2 *Passive Replication*

This technique is a *primary-backup* approach [4], where one replica handles the client's requests and replies. The *primary* replica is the one responsible for the request processing, while the other replicas act as backups. Therefore, the *primary* replica has a special role in comparison to the *backup* ones since it is the only replica that processes requests.

Passive replication depends on a reliable multicast protocol to send the updates processed by the primary replica to all backups in the system (e.g., *View synchronous multicast*

(VSCAST)). Since *backup* replicas solely apply updates and not process them, the determinism constraint is not required by this mechanism. The VSCAST protocol ensures that, when the primary replica sends an update, it is received by all replicas in the system, and in the same order. Transaction order is set by the primary replica FIFO processing and respective communication channels, while the delivery guarantees are granted by VSCAST mechanism, enforcing that either all replicas receive the updates or a new view is implemented. When the primary replica receives the update sent, this implies that all replicas also received the message. Otherwise, if the update is not delivered to one replica, neither one receives it and a new view gets implemented with the operational replicas present in the group membership, multicasting the update message again. The communication primitive VSCAST is inserted within the abstraction of group membership, allowing to apply views with a well-defined set of operational replicas.

The core function of the algorithm described in Figure 2, assumes no failures. Therefore, clients issue requests to the primary replica, which executes the operation. After a successful execution, the updates are transmitted to the backup replicas using VSCAST mechanisms. Following the application of those updates, all the backup replicas must send their acknowledgment to the primary, that ultimately replies to the client.

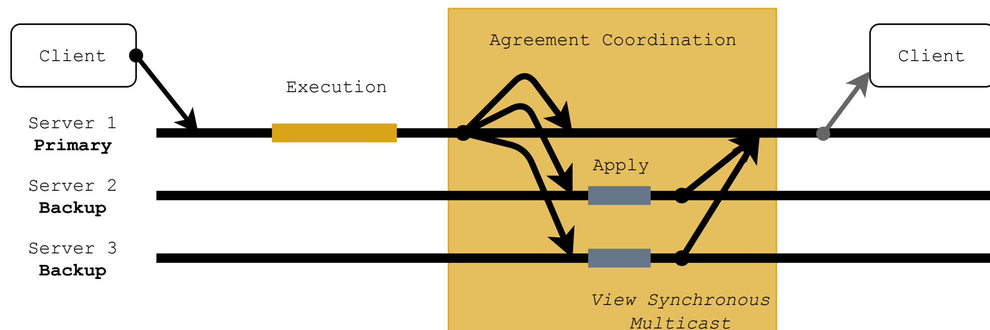


Figure 2: Passive replication mechanism.

If the primary replica experiences a failure before sending the updates to the backup replicas, the client will get a time-out and must reissue the request to the new primary server, after learning its identity. The primary server will then process the request again. In the event of the primary replica failing after sending the update message, when client reissues the request, it gets an immediate response back because the new primary typically already applied the updates related to that request.

The Passive replication algorithm in its core allows non-deterministic servers (i.e., multithreaded servers) [26]. This is possible as there is one single replica executing transactions (i.e., primary replica), while other replicas merely apply deterministic updates. This mechanism requires less processing power compared to active replication, considering that backup replicas are passive machines that apply updates with possibly low resource consumption,

while primary replica interacts with clients and processes all requests, requiring a large amount of processing power and being susceptible to saturation.

However, there are some disadvantages, as a transaction might need reissuing in case of a failure. A reissue can occur if the primary fails before replying to the client. After timing-out the client must learn the primary replica's new identity to reissue the requests, meaning that failures are not masked.

Moreover, passive replication does not allow replicas to execute irrevocable operations. That is, operations whose effects cannot roll-back (e.g., local system calls) since some of the backups can fail to apply the updates and the transaction can abort (Agreement Coordination Phase, Wiesmann et al. [26, p. 3]).

2.1.3 *Semi-Active Replication*

Semi-active replication protocols focus in mitigating the non-determinism restriction of the active replication technique [20], extending the algorithm by allowing non-deterministic operations to execute.

This protocol introduced the terms *leader* and *follower*, related to *primary-backup* approach to allow non-deterministic operations to execute in the system.

The mechanism executes operations in a similar way as the active technique, where client requests are sent towards a server group as opposed to a single replica, and the whole system executes the transactions, replying to the client accordingly.

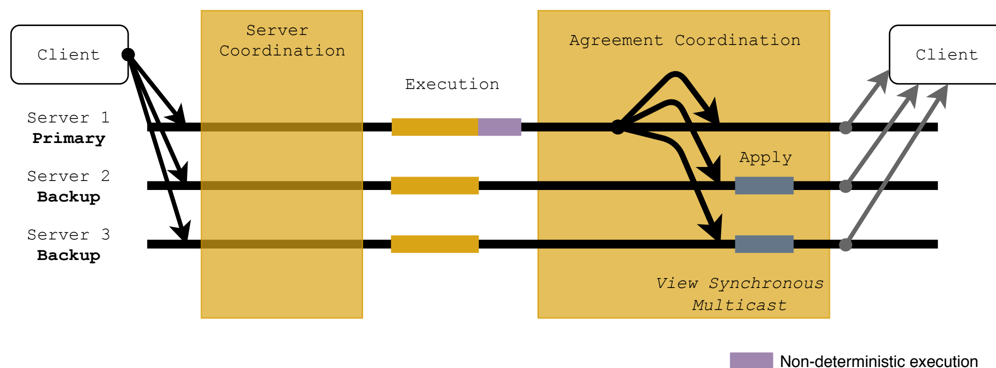


Figure 3: Semi-Active replication mechanism.

The overall execution of a request in the system is illustrated in Figure 3. The main replication algorithm is detailed in the image, allowing to understand the similarities with the active replication. Only the non-deterministic execution differs from the base line algorithm.

The semi-active replication mechanism, is very similar to both Active and Passive replication. When a client issues mixed operations (i.e., request containing both deterministic and

non-deterministic operations), this algorithm uses the full potential. If an operation containing both deterministic and non-deterministic transactions arrives, servers coordinate similarly to active replication and, all replicas execute the deterministic operations, waiting while the non-deterministic ones are executing in the leader replica. Afterwards, using the VSCAST primitive the updates are multicasted to all replicas and applied by the followers, replying to the client after the execution of those updates.

For every non-deterministic transaction, the *Execution* and *Agreement Coordination* phases are reproduced, increasing latency and making it the only drawback of the mechanism.

Although there is a considerable disadvantage to this approach, there are also benefits, namely: abstracting the client from failures and, having a leader to enable the execution of non-deterministic operations.

This technique also introduces a lower overhead than passive replication, considering the size of the update messages forwarded to the followers. Since part of the request was already executed by the followers alongside with the leader, the updates sent to the slaves are the ones related to the non-deterministic operations within the request, consequently, being smaller.

2.1.4 *Semi-Passive Replication*

The semi-passive replication is a variant of passive replication, without requiring a group membership to agree on a primary replica. This technique uses, as an alternative, a consensus protocol to achieve the goal of electing a primary replica in each round [10].

Semi-passive replication is based on a rotating coordinator paradigm, also used to solve the *consensus* problem [6] to decide on an initial value for a request. This technique relies on *Consensus with Deferred Initial Values (DIVconsensus)* to choose the primary replica. That replica gets selected during each round of consensus. In fact, the primary replica is the coordinator elected which holds the responsibility to process the request.

Consensus is one of distributed system's fundamental problems, that requires a set of processes to agree on a decision (e.g., a leader election or the order of received requests [6]). A consensus protocol must be fault-tolerant and reliable considering that processes within the set can fail or become unreliable. Even if there is only a suspicion of failure, the consensus protocol must be capable of obtaining a single consensus result from all the processes, obtaining a quorum (i.e., the minimum number of votes to achieve a decision) from processes.

The rotating coordinator paradigm [6] allows two different timers, one aggressive timer to rotate the server, allowing a fast reaction to any failure and another conservative timer to effectively exclude the failing server from the group. This mechanism allows a fast reaction to failures, without the cost of excluding a server if there is an incorrect suspicion. In

contrast to the passive replication protocol, this approach does not require the client to know the identity of the primary replica, since requests are sent to all replicas similarly to active replication, meaning that the protocol abstracts client crashes, considering that all the replicas send a reply to the client.

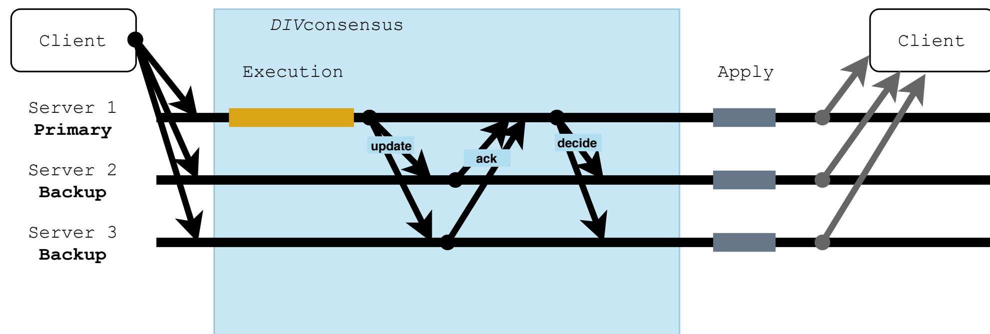


Figure 4: Semi-Passive replication mechanism.

The generic implementation of this technique is illustrated in Figure 4, assuming no failures. The client must send the request to all servers then, the *DIV*consensus is solved electing a coordinator (i.e., the primary server). The primary server, similarly to the passive replication has to execute the transaction and send the updates to all backup replicas.

Semi-passive mechanism retains some characteristics from passive replication. As such, in the absence of a failure, only the primary replica processes the requests meaning that executions do not need to be deterministic, maintaining those advantages from the passive technique. Those updates are sent using the consensus algorithm during communication, multicasting the update messages and waiting for the majority of the acknowledgments. Afterwards, the coordinator decides and sends the decision to backup replicas. As soon as the decide message arrives, all servers apply the updates and reply to the client.

If the primary server is ever suspected to have experienced a failure, a new round of the *DIV*consensus gets executed, electing a new coordinator that will be the primary server for that round. After the election, the primary server waits for an estimate message from the majority of servers, as these messages can contain the initial value that might be used by the new primary server to avoid processing the requests again. In the worst case scenario, the new server needs to repeat the processing and reply to the client.

The semi-passive replication allows non-deterministic servers. Therefore, this technique fully supports multiple threads, and replica failures are hidden from the client. Semi-passive protocol combines both advantages of active and passive replication, although having limitations when the primary server fails, increasing the response time to the client.

2.1.5 *Hybrid Transactional Replication*

The *Hybrid transactional replication (HTR)* [16] mechanism is a protocol that, implements both active and passive replication, deploying an oracle to perform decisions on the replication technique to apply. This oracle decides for each transaction, the replication technique employed, being either a State Machine (i.e., active replication) method or, a Deferred Update (i.e., passive replication) using a multi-master approach.

The multi-master approach applied to the passive replication mechanism, allows a client to issue requests to all machines, as opposed to the single primary in passive replication. With a multi-master approach, any machine that receives requests, processes the operations within those requests, as others solely apply the updates, showcasing a dynamic primary replica. This implementation is common in database specific protocols (e.g., Postgres-R), described in Section 2.2.

This mechanism uses an oracle to make decisions, being tuned with various inputs, namely, network saturation, system load or the performance of garbage collection. All data collected is fed to a machine learning algorithm to tune the oracle, allowing conclusions to be assertive and effectively producing decisions about the replication technique to use when executing a transaction.

The core of the HTR algorithm is the primary-backup technique, extended with a state-machine approach. When a transaction arrives at a replica, it is fed to the oracle, deciding whether that transaction gets executed in the *State-Machine (SM)* mode or *Deferred-Update (DU)* mode. Whenever a transaction gets performed in the DU mode, the receiving thread is responsible for the processing concerning the transaction, raising the commit and then, waiting for certification from other replicas. The process is different as the SM mode is chosen. In this method, the transaction gets broadcasted to all replicas and, it must execute in the main thread of those replicas. Besides executing SM operations, the main thread also conducts the certification tests for the DU mode updates. The main thread is vital to guarantee consistency in this mechanism as it ensures that only one SM transaction can execute at a time and no certifications are performed concurrently, hence delaying the certification of DU operations until the main thread ends the SM execution.

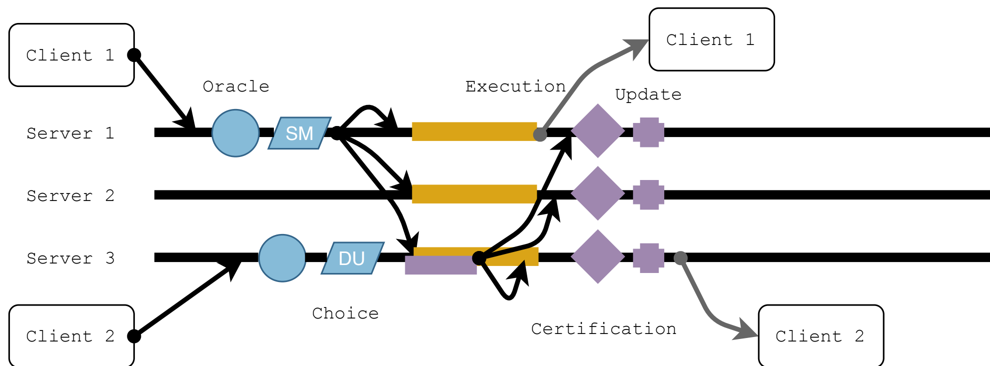


Figure 5: Hybrid transactional replication mechanism.

In what regards the concurrent execution of DU alongside with SM requests, there are no guarantees that both commit, as they can execute concurrently. Although DU and SM operations can execute concurrently in different instances, both DU certifications and SM transactions get executed sequentially.

Figure 5 depicts an abort-free execution of both a DU and SM transaction that is replicated across both machines. As depicted, the DU operation executes concurrently with the SM one, but is only certified after the SM execution finishes. Moreover, in this protocol when a transaction executes irrevocable operations, it is executed in the SM mode exclusively, guaranteeing an abort-free execution.

Since DU operations can execute concurrently with SM transactions, when both conflict, if the DU operation is certified after the SM execution, the DU operation is aborted.

2.2 DATABASE SPECIFIC REPLICATION PROTOCOLS

Databases benefit from specific replication techniques that allow high throughput and dependability.

In contrast with the prior techniques, database specific protocols have a special focus on data, as such, the architecture behind these protocols allows a better insight into the replicated data and their conflicts, as opposed to the generic approaches. Moreover, both approaches addressed below support their implementation on multi-master passive replication mechanism with an update-anywhere approach, enabling flexibility and higher computation performance of requests, as required by database systems.

2.2.1 *Conservative execution*

Conservative execution in a database system is based on conflict classes of transactions. Those transactions only commit when all replicas confirm that no concurrent requests con-

flict with each other. Therefore, this protocol guarantees null abort rates as every operation passes through a verification before the execution.

The conflict-free execution of the protocol is illustrated in Figure 6, where two requests execute concurrently and, since the transactions have different conflict classes, after verification all replicas install updates concurrently has shown in the illustration. Each replica multicasts the request to all sites in the group. However, only the master site where the request is issued executes it and, after completion a message is sent with the updates to be applied.

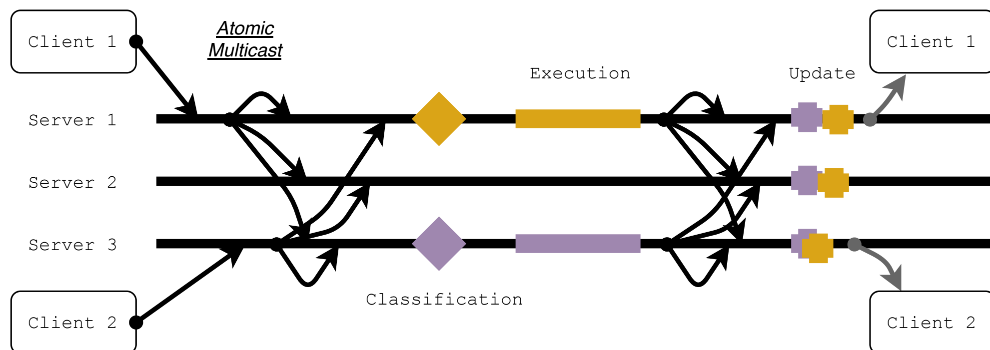


Figure 6: Database conservative replication.

When conflicting operations are detected, both are executed sequentially. The sequential execution is allowed considering that each site has a queue for conflict classes. Therefore, if a transaction touches a combination of conflict classes, every site adds the transaction to the queue of each class, allowing fairness and, executing conflicting transactions sequentially. If there are operations that eventually do not conflict, they can be executed concurrently and are applied as such, similarly to the example discussed.

The conflict classes are a method to identify if two transactions conflict with each other. Before executing a transaction they are labeled according to their set of conflict classes, since there are different conflict classes with more or less precision.

A coarse grain conflict class is a less precise category that can be as extensive as a full table. Differently, a fine grain conflict class is a precise one, and is typically defined as rows or fields. However, fine grain classes are frequently more difficult to achieve. Although the most suitable conflict class is the fine grain one, it is difficult to achieve such categorization. Accordingly, conservative execution can be performance-wise expensive to implement, specially with large queries accessing numerous tables and rows.

Therefore, the number of transactions executed sequentially can be a bottleneck to the system, since it solely depends on the conflict class categorization. A set of fine grain conflict classes can lead to a high number of operations executing concurrently. However, a coarse grain leads to a majority of transactions executing sequentially, as the classification will detect that they touch the same data.

2.2.2 *Optimistic execution*

Considering an optimistic execution, all replicas execute concurrent requests independently. Only after the execution, the write-set or conflict classes of the transaction are sent to the other replicas to check for conflicts and, if there is a conflict, the transaction is aborted and retried. Otherwise, the transaction is committed and, successfully replicated.

Executing requests fully optimistically can lead to a high abort rate and, typically those aborts increase with the number of requests issued. Both mechanisms described in this section share some features, as both protocols use a multi-master replication approach, receiving and executing requests without any prior coordination. After execution, there are two approaches to certify executed transactions and, both approaches diverge in the implementation of the termination protocol.

2.2.2.1 *Postgres-R*

In Postgres-R [15], transaction's read-set is not sent to other replicas. Consequently, only the executing replica can certify it. When the replica executing the request receives a commit, it multicasts the write-set and transaction id. As soon as the executing replica finishes processing prior transactions, it certifies the current one. After the certification process, the replica either multicasts the updates to other replicas, or aborts the operation.

This behavior is illustrated in Figure 7, where two requests are executing concurrently, and it is possible to depict *Server 3* waiting for the processing of the first transaction on step **2**, before certifying next one and broadcast the updates. Meanwhile the *Server 1* certifies the yellow transaction, as there are no transactions issued before on step **1**. Once certified, the *Server 3* broadcasts the updates, as illustrated on step **3**, since no conflicts occurred on both requests.

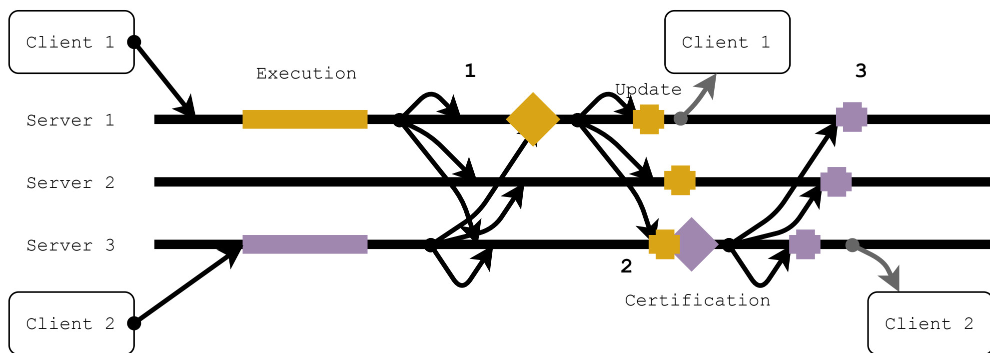


Figure 7: Database optimistic PostgresR replication.

An abort-free execution is illustrated in Figure 7, and when a conflict is detected by this termination mechanism, the transaction is aborted and the updates are not applied.

2.2.2.2 *DBSM*

The *Database State Machine (DBSM)* replication protocol [18] handles the certify process differently from Postgres-R. In this protocol the read-set is also propagated towards other replicas, so that every instance can certify transactions.

When a commit is issued by a replica, it multicasts to other replicas the id of the transaction, the version of the database where the transaction was executed, the write-set, and read-set. Once the multicasted message reaches other replicas it is orderly processed. Each replica certifies the transaction by verifying if there is an intersection between the read-set and write-set of the current transaction with others that have been committed after the fixed view (or version) of the database. If there is no intersection, the transaction commits by applying the write-set to the database version multicasted, otherwise the transaction aborts.

Figure 8 illustrates the execution of two requests considering a DBSM replication approach. The issued requests execute concurrently and apply the aforementioned termination protocol. Since the illustrated transactions do not conflict, both commit and, after the certification, the updates get applied in all replicas, ultimately maintaining a coherent and replicated state.

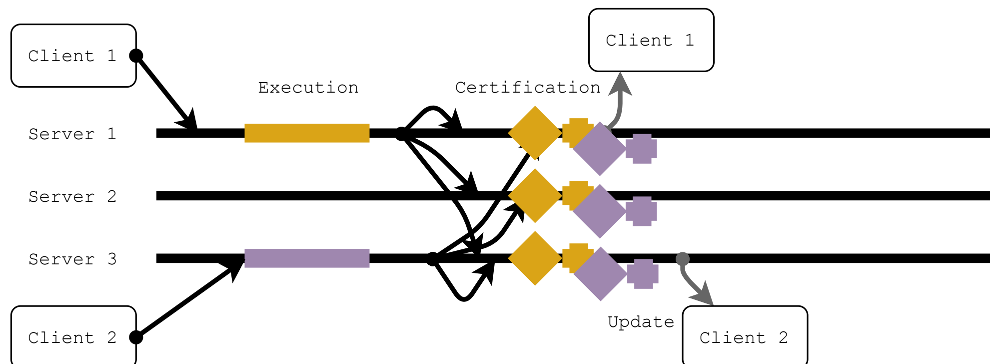


Figure 8: Database optimistic DBSM replication.

Similar to Postgres-R, this Figure illustrates an abort-free execution and, in this termination protocol if a conflict is detected, the transaction aborts and the response is transmitted to the client.

2.2.3 *Hybrid execution*

The hybrid execution technique was developed in *Akara* [7], where the replication algorithm has its core on a hybrid solution that sits between the conservative and the optimistic execution. The core objective of this solution is to offer the best of both prior replication tech-

niques, enforcing a conservative execution to ensure fairness, efficiently manage resources by using an optimistic execution, and ultimately provide active replication on demand.

In this technique, issued requests are totally ordered and classified prior to any execution, similarly to the conservative execution. After ordering and classifying the requests, they are placed into a queue to be processed. Requests selected from the queue are executed optimistically. Therefore, transactions might abort due to conflicts with concurrent transactions.

To select requests for processing an admission control mechanism is deployed. This mechanism implements a policy that ensures the execution of n concurrent requests, attempting to exploit the idleness of the servers, by reducing the contention of the conservative execution. The admission control deployed does not allow executing only non-conflicting transactions, as that would make this technique fully conservative.

After being picked from the queue and, ultimately executed, if the transaction is ready to commit, its changes are propagated to all other replicas and the transaction commits. However, if a conflict is detected, the transaction is re-executed conservatively. The conservative execution ensures a high priority execution of the transactions over any local executing transactions. This protocol also allows transactions to run actively, hence making possible not only to replicate transactions but, also to replicate *Data Definition Language (DDL)* statements.

3

MIDDLEWARE-BASED REPLICATION MECHANISM

3.1 DATABASE REPLICATION SYSTEMS

As different replication protocols and techniques exist to fulfill the needs of the applications, also different software and services offer such implementations as a product. The replication software can be embedded as part of an operational database or offered as *Software as a Service (SaaS)*.

These services are in some aspects related to this dissertation purpose, either relying on the same basis or by using the same principles of replication.

3.1.1 *PostgreSQL Logical Replication*

The PostgreSQL logical replication is enabled through the usage of a *Write-Ahead Log (WAL)*, streaming the changes (i.e., write-sets) through the network to as many replicas as configured in the system. A WAL is a persistent log written before any commit operation to the database and, is typically used for database recovery. PostgreSQL logical replication mechanism follows a *Publisher-Subscriber (Pub-Sub)* paradigm, where the publisher is the primary replica, and subscribers are backup replicas. Committed transactions are written in the WAL together with a message published to all subscribers containing the updates to execute. To receive updates, subscribers must connect to replication slots. Those slots are created by the master to publish the write-sets of the transactions.

Since logical replication provides fine-grained control over both data replication and security, replication slots maintain independent logs to the subscribers. When a backup replica that is too fast applying updates and, another is too slow, the transactions provided to the slow replica will not be affected by the fast one, since every replica consumes updates from a different slot as depicted in Figure 9. These slots have different capabilities and, plug-ins can manipulate slots allowing to retrieve write-sets without configuring any replication.

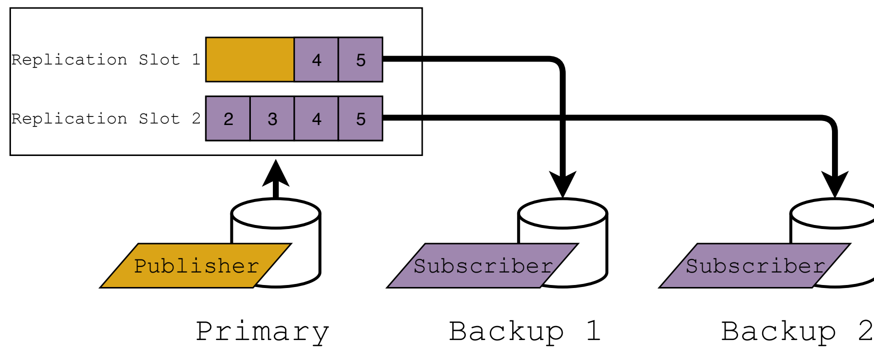


Figure 9: PostgreSQL logical replication slots.

There are two main variations of logical replication, namely: the synchronous and asynchronous replication. Although both approaches have advantages and disadvantages, only the synchronous replication is reliable enough, considering that it is the only alternative that guarantees that data is safely replicated in more than one replica, ensuring availability.

In the asynchronous replication, when the database receives a transaction, the execution is identical to the single replica approach, the only core difference is that once in durable storage (i.e., in disk), the transaction is streamed as a write-set to other replicas. That stream uses the logical replication slots to send the write-sets, as illustrated in Figure 9. With this replication technique, there are no guarantees that when the client receives a response, the transaction is already in two different locations (i.e., two copies). Therefore, in contrast to the synchronous replication, there is a probability that, if a replica fails, an already executed transaction is lost.

When implementing PostgreSQL native synchronous replication, it is possible to set the dependability required. This configuration can offer different guarantees concerning replication, depending on the value set. There are five possible values to configure the native replication, namely: *on*, *remote_apply*, *remote_write*, *local* and *off*.

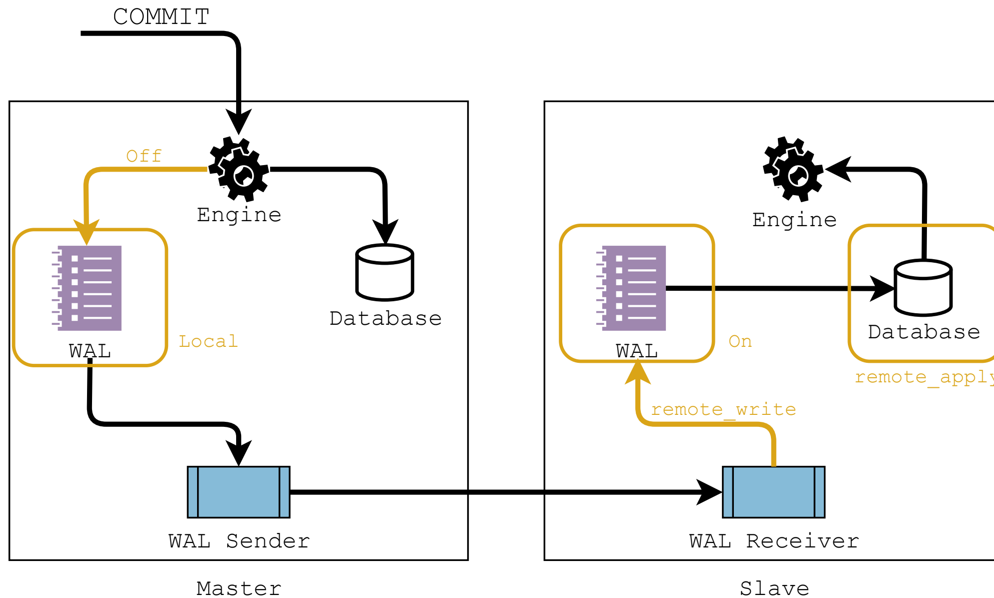


Figure 10: PostgreSQL *synchronous_commit* configuration values.

As illustrated in Figure 10, different configurations offer different tolerance to failures, and consequently, different throughput values. The default and, most advantageous value is *on*, ensuring that, when the master replica receives a transaction, it will not reply to the client until the backup replica has safely written that transaction to durable storage. Therefore, if a failure happens, the transaction will not be lost. However, when the *synchronous_commit* field is set to *off*, the master replica will not wait until the transaction is safely written on WAL, providing no guarantees whatsoever that the slave will receive the write-sets if a failure happens. Even if the client retrieves a successful response, the transaction might not be replicated onto the slave replica.

Additionally, transitional values are also available, providing distinct features. The *local* value guarantees that the transaction will be written in the master replica's WAL (i.e., durable storage), but gives no guarantee whatsoever regarding persistency onto the slave one.

Both *remote_apply* and *remote_write* are values that influence operations on the backup replica. As depicted in Figure 10, the *remote_write* value will force the master replica to wait until updates are written to the operating system on the backup replica but, not necessarily in the disk and, the *remote_apply* value waits for the confirmation that transactions are effectively applied in the backup database, causing the transaction to be read-ready on the backup machine. Typically, the *remote_apply* is the safest value, since when the client receives a reply it means that the transaction can be observed in the backup replica. However, those guarantees are not necessary, since the *on* value already persists the operations on the backup's durable storage.

3.1.2 *Amazon Aurora*

The Amazon Aurora database is a log-based mechanism allowing high throughput, resilience, and scalability [25]. This technological advance in Amazon Aurora began with the identification of fundamental points that influenced the system bottlenecks. The developing team concluded that both the caching and logging system of current relational databases are outdated. Therefore, they focused on both layers, having as a developing core the MySQL Community Database Engine.

In Amazon Aurora, it is stated that *the log is the database*, relying on a high-throughput log to maintain a coherent state between replicas. In contrast to a typical database that writes full pages to durable storage, Amazon Aurora solely writes redo logs to storage since they have a smaller size. This approach is effectively a significant and thoughtful improvement to the current solutions since a typical relational database only requires WAL logs to be written in durable storage.

The setup of this approach is defined as a primary-backup solution, where the primary replica only writes the log records to the storage service, alongside with metadata to other instances. Accordingly, Amazon Aurora relies on the redo-logs written on disk, more specifically the *deltas*, instead of full pages. This allows to produce a database after-image, using the before-image by applying those deltas. Consequently, this means that Amazon Aurora replicas use the redo-log written to durable storage to apply the changes to their buffer caches, maintaining a coherent state across the cluster.

The redo-log is written to a fleet of storage nodes and, in order to tolerate failures, a quorum based protocol is used. A quorum is a minimum number of votes that enables the verification that a majority of replicas have written the redo-log. Moreover, different quorum configurations can be used, depending on the fault-tolerant guarantees needed [25].

To guarantee the order in the log, an *Log Sequence Number (LSN)* is used as a monotonic value that keeps increasing and, additionally, a *Volume Complete LSN (VCL)* is also applied, denoting the highest sequence number to which the database can guarantee availability. When committing a transaction, a worker thread records the commit LSN. As the VCL advances, the database identifies the transactions waiting to be committed and uses a dedicated thread to reply to the client. Effectively, when writing the deltas to the volume fleet, there is the guarantee of a fully distributed write. Giving the same guarantee than the *on* parameter used in the PostgreSQL native replication, depending on the setup.

As detailed Amazon Aurora database relies on a log to achieve high throughput and dependability across different regions. Moreover, in Amazon Aurora, all the replication stack is based on services offered by Amazon itself. Therefore, this database is offered as a final production solution and, it is a part of their portfolio, available as a SaaS.

3.2 DISTRIBUTED LOGS

Distributed logs are an efficient and high-performance solution to enable replication and replica communication. Distributed logs remain spread throughout different machines, maintaining ordered events and messages, ultimately offering a compelling abstraction of the whole distribution.

One of the most high-performance tools that manage distributed logs enabling stream processing, high-availability, strong coherence, and durability is Apache's DistributedLog API, supported by their log segment store, Apache BookKeeper.

The Apache DistributedLog is a *Application Programming Interface (API)* provided by Apache BookKeper [2], offering distributed logs with replication across different sites. Each site implements one replica of BookKeeper cluster (i.e., a bookie), offering durability, replication and strong consistency.

The DistributedLog acts as a high-level API to BookKeeper log stores and, therefore, BookKeeper is essential to provide the log storage and fault-tolerant mechanisms.

From a top-down point of view, Apache BookKeeper offers log stores (i.e., ledgers) stored in individual servers called bookies. A bookie is an individual BookKeeper storage server. Bookie servers maintain fragments of ledgers, not entire ledgers, providing more performance, both to reads and writes. When entries are written to a ledger, they are split across an ensemble (i.e., to a sub-group of bookies) rather than to all bookies.

To guarantee strong consistency, when adding entries to a ledger, BookKeeper relies on a two-phase commit variation, the *LastAddConfirmed (LAC)*. This mechanism allows writers to push a batch of entries incrementing a counter designated as *LastAddPushed (LAP)*, receiving acknowledgments in order when they are effectively committed, increasing the LAC pointer.

The BookKeepers metadata storage and handling are delegated to Apache ZooKeeper [3], entrusting the responsibility for storing information related to ledgers such as, availability, state and location. Apache ZooKeeper is able to maintain correct and updated metadata regarding the bookie's availability using ephemeral *znodes*. The use of these nodes ensures that they only exist as long as the session that created them is active, allowing availability tracking. Similar to Apache BookKeeper ledgers, the metadata stored in ZooKeeper is handled by a quorum of replicated servers, maintaining the metadata available even in the event of failures.

Overall, Apache BookKeeper and Apache ZooKeeper are designed to be reliable and resilient to failures, consequently the DistributedLog API also benefits from those guarantees.

3.2.1 *Architecture*

The DistributedLog core architecture focuses on the implementation of log streams. Those streams are an ordered and immutable sequence of log records, written into log segments. Although applications get a continuous sequence of log records from a stream, those records are physically stored as multiple segments that have the same replication configuration, being allocated, distributed and stored in a log segment store.

This architecture achieves consistency with the employment of bookkeeper's LAC protocol that is a variation of the two-phase commit. The architecture relies on that protocol to build its data pipeline. When a writer appends an entry to the log, a LAP pointer is used. This pointer is the entry id of the last batched entry pushed to the log segment store. Log entries can be written out of order but, acknowledgments are required to be performed by the entry id order, indicating that LAC is the entry id of the last record acknowledged. Both pointers described help guarantee data consistency, similar to the two-phase-commit protocol, ensuring that solely committed and verified data is visible to readers. Therefore, to achieve those guarantees, entries between LAC and LAP are not visible to readers, as it is unacknowledged data. Exclusive data from LAC pointer backward is visible to readers since it is already verified and committed as illustrated in Figure 11.

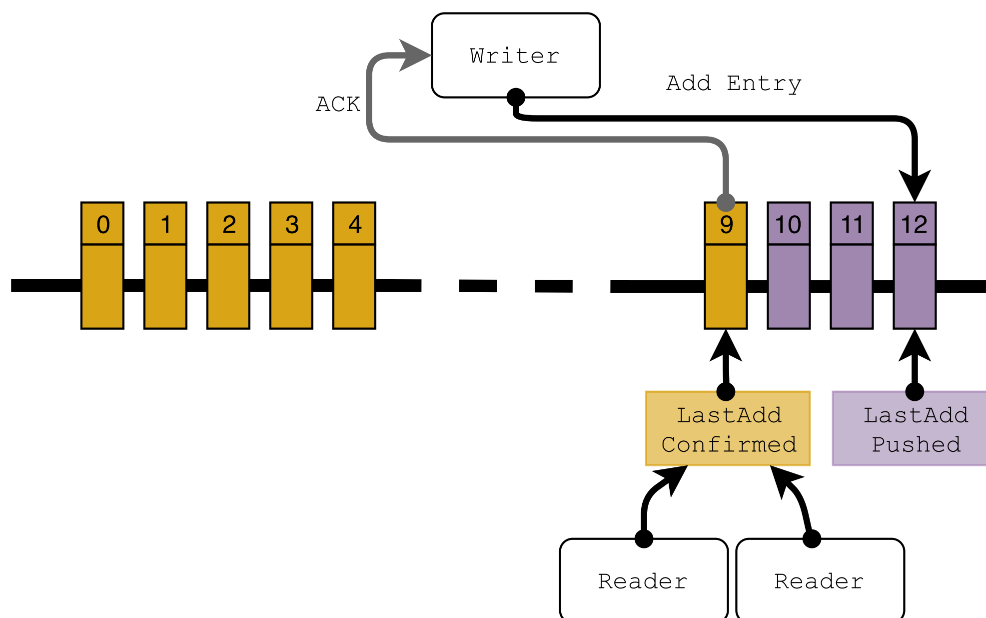


Figure 11: Last Add Confirmed Mechanism.

Although the LAC is a useful mechanism to guarantee consistency across readers, it is not suitable to guarantee correctness when the writing ownership of a log stream changes. For that reason, a fencing mechanism is built into bookkeeper to achieve data consistency across multiple writers when network partitions occur. By fencing data in the log seg-

ment store and conditionally updating log segment metadata (using a versioned set) in the metadata store, network partitions and failures are tolerated. When a log stream changes ownership, the new client fences the log segment to guarantee that no more records are pushed, consequently, the unreliable old owner (e.g., network partitioned) no longer can write to the log. Ultimately, the new log owner reclaims the log segment and continues to push records to the log, considering that it currently holds the ownership.

Alongside with the fencing mechanism, an ownership tracking is used to manage the log stream control, opposed to the typical leader election. This ownership tracking guarantees correctness by using zookeeper's ephemeral *znodes* to track the ownership of log streams, as zookeeper already provides sessions that can be used to track leases for failure detection.

The DistributedLog API uses two roles in its architecture, writers, and readers. These parts hold the responsibility to push and read records from the log, respectively. The writers and readers implementations are described in the following sections 3.2.2 and 3.2.3.

3.2.2 *Log Writers*

The DistributedLog has two implementations of writers. The first alternative is a fat client that utilizes a direct API to interact with bookkeeper and zookeeper. By using this client, write locks and streams ownership are handled by the application. Therefore, this method is recommended to applications that require write-ordering across different replicas and, applications that already deploy an ownership management system.

The second approach is the use of a thin client, relying on a stateless write-proxy. The write-proxy allows sequencing writes from many clients, managing the ownership of log streams, forwarding writes requests to storage and, handling load balancing alongside with fail-over. However, write-proxies only guarantee read-ordering.

The thin client is the approach used, considering that developing a distributed lock mechanism would be too expensive. Therefore, as the development aims to build a multi-master protocol with high throughput, the thin client and, write-proxies is the most suitable alternative.

3.2.2.1 *Write-proxies*

A write-proxy is a deployed server that accepts fan-in requests from publishers and writes those requests to the streams. A write-proxy manages streams, redirects requests and, tracks ownership changes. Typically write-proxies are high-throughput servers that can be deployed anywhere in the world and, as many as possible. Those write-proxies maintain a membership system, allowing them to redirect requests and receive notifications when membership changes. Therefore, a write-proxy can take the ownership of a log stream when some other fails, using the aforementioned fencing mechanism. Moreover, a deter-

ministic routing algorithm allows multiple clients to interact with the other write-proxies when the current stream owner proxy is unavailable.

The write-proxy membership is maintained by the Apache ZooKeeper, similarly to bookies. Consequently, it can provide another layer of fault-tolerance to the whole system design, maintaining the access to log streams always available considering that the group is self-managed and handles changes in the ownership when failures and network partitions occur.

3.2.3 *Log Readers*

In what concerns the readers implementation in the architecture, considering that they merely read committed records and act as followers, read-proxies do not have to track ownership. Moreover, to route readers to the respective read-proxies, a consistent hashing routing mechanism is deployed.

To be able to read records from the logs, readers have to position a pointer based on either the *DistributedLog Sequence Number (DLSN)* or Transaction ID. These pointers are used by the reader to retrieve records continuously until reaching the end of the log. When the reader has caught up with the writer, it waits for notifications regarding new records. Readers can catch up with the writer by reading LAC records.

To read specific entries from the log and, ensuring low latency even in the presence of failures, a speculative read operation is used. This operation consists of sending a read request to the first replica and, if the client does not obtain a response within a speculative time-out (i.e., before experiencing an actual time-out), it sends another read request to the second replica, promptly waiting for both responses (either the first or the second response). This process continues until it receives a valid reply. Moreover, if no response is received, the client raises a time-out.

A combination of both operations allows the reading of the next available entry in the log segment. In this operation, a client sends a long poll read request along with the next read entry id. If the log segment store already owns the committed entry, immediately returns with the latest LAC, alongside with the requested entry. Otherwise, it waits for the LAC to reach the requested entry id and sends the response back with the requested entry.

The log readers implementation enables the recovery mechanism of the hybrid middleware. When a replica requires recovery from a crash, it can open the log stream at a given DLSN or Transaction ID, and execute the missing transactions. Moreover, when a (partial) system failure occurs, operational replicas can re-open the log stream that is the failing replica's responsibility, continuing the execution of transactions. This mechanism is developed concurrently, being in the scope of the same project but, outside the scope of this thesis.

3.2.4 *Log Streams Replication*

As previously mentioned, the Apache BookKeeper alongside with the DistributedLog API provides fault tolerance guarantees, durable storage and, high throughput. Considering that the developed middleware relies those guarantees, it is imperative to understand how the replication protocol works.

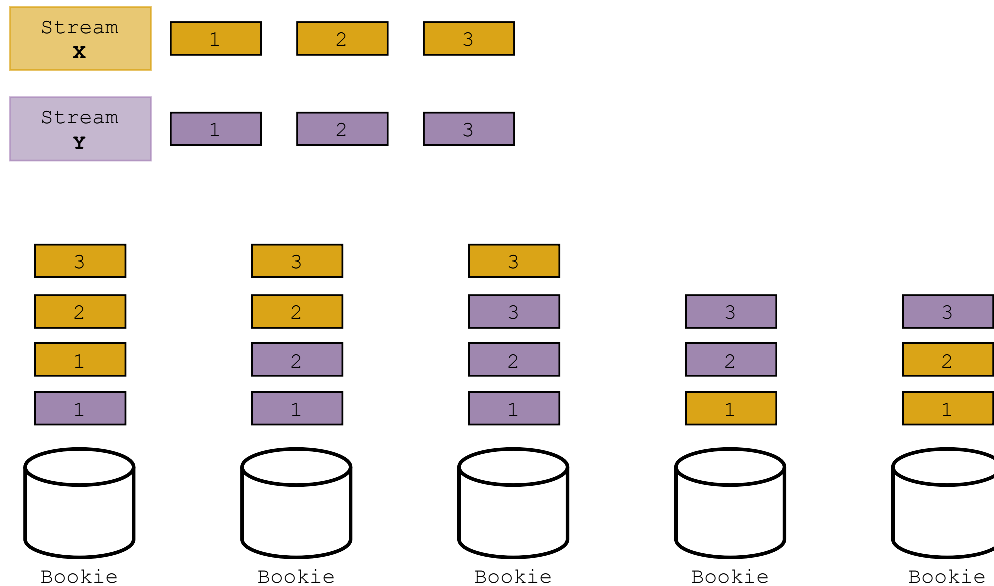


Figure 12: Apache BookKeeper log segment allocation.

Figure 12 denotes the Apache BookKeeper's allocation of log segments. It is important to note that, a log stream is split into various log segments, and these segments are respectively replicated through different bookies. The streams distribution into multiple log segments allows an uniform allocation across multiple storage nodes for load balancing. Moreover, this distribution improves the overall system balancing in both read and write requests.

As depicted in the Figure 12, a five nodes ensemble is adopted to illustrate the load balancing. With two streams split into three segments, it is conceivable to illustrate the balancing within the bookie cluster. Expressed above is a three write quorum ensemble with five replicas, denoting that a segment must be written in three copies. Moreover, the Apache BookKeeper will employ a judicious placement to improve the throughput, read performance, and avoid the saturation of bookies. Unmistakably, with three nodes, the bookie segment allocation is trivial to picture since all segments are written in every node.

Under a more technological specific focus, the replication relies on a rack-aware placement policy. This policy guarantees that all replicas maintaining the same log segment are allocated in different racks, ensuring network fault-tolerance. Moreover, in addition to the rack-aware policy, a region-aware placement policy for a global replicated log is also

available. This placement policy ensures that the same log segment is allocated in multiple data-centres. Therefore, it requires acknowledgments of a majority of those data-centres spread replicas. The acknowledgment in this policy requires a parameter that ensures that the quorum covers at least a minimum of regions to guarantee that the system is kept operational without loss of availability.

Ultimately, this policy can be used in conjunction with the rack-aware policy to distribute data uniformly across regions, within the same region and across different racks, providing high-availability and resilience.

3.3 DISCUSSION

Database replication protocols are moving towards a high level approach in what regards data replication. Protocols are maturing to have more abstraction and enhance data independence, without compromising neither availability nor data coherence. Moreover, an immutable sequence of log messages can help achieve such purpose, offering both performance, availability, correctness and high-level abstraction of replication.

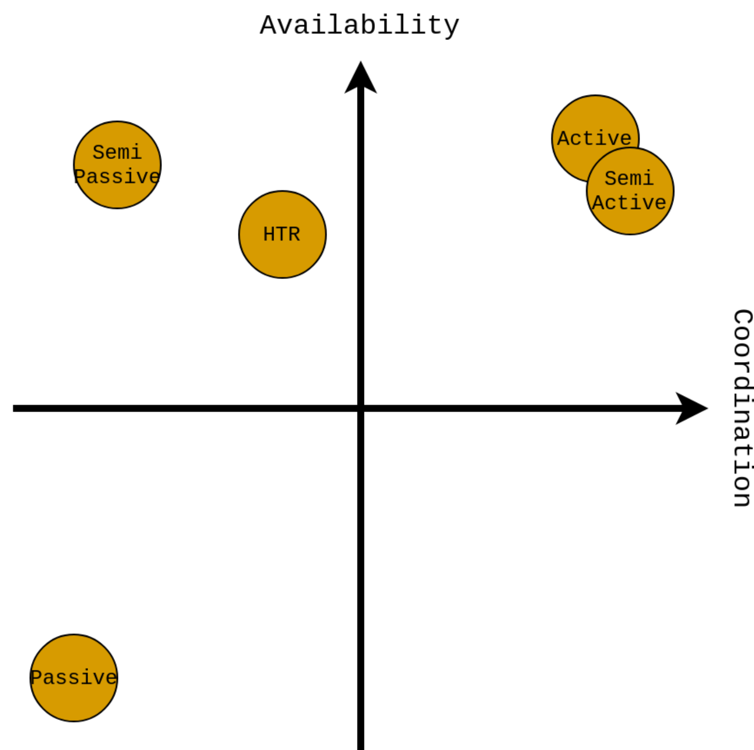


Figure 13: Protocols comparison within a quadrant.

Figure 13 enables an insight of where all the protocols and techniques fit into, in a quadrant field plot representation. Both axis characterize important features, namely: availabil-

ity and server coordination, providing insight into two important properties of replication protocols. Furthermore, the generic replication techniques are arranged in the quadrants according to their compliance with the features described in the axis.

As observed, both active and semi-active replication protocols rely heavily on server coordination to guarantee high availability. With prior server coordination, both protocols ensure that no service disruption nor time-outs occur in the presence of failures. On the opposite side of the system, the passive replication protocol is illustrated. This protocol does not depend on any server coordination but lacks availability. Consequently, when a failure occurs in the primary replica, the client suffers a time-out and, there is an overhead to select a new primary replica (i.e., execute a leader election).

Therefore, passive replication can experience performance issues on the primary replica, and the active replication has the drawback of the determinism required in the whole system. Also, semi-active and semi-passive replication have limitations of their own. Some of those limitations are mitigated by database replication protocols (e.g., using multi-master). However, database replication protocols do not enable a simple separation between replication logic and data, denoting a complex implementation and maintenance.

The second quadrant of the system is a perfect position, as there is availability and limited server coordination to achieve it. As illustrated in Figure 13, there are two protocols that fulfill both features, offering high-availability with limited server coordination. Semi-passive replication relies on the consensus protocol to achieve such a position, not requiring strict server coordination, yet enabling failure abstraction. Moreover, the HTR protocol also surpasses those limitations with the usage of an oracle and, a multi-master approach.

As depicted, there is space to endeavor high availability without relying on server coordination since semi-passive replication does not implement a multi-master approach. Therefore, the aim is to develop a hybrid middleware replication protocol with database-specific insights, without being tightly merged with the data. Both replication services described in this chapter have clear limitations. As of PostgreSQL Native Replication, the most strict limitation is the required usage of the data source. Therefore, the replication service can only be used within the PostgreSQL database.

Differently, Amazon Aurora is similar to the proposed hybrid middleware solution, as it allows the integration with different data sources, namely: MySQL and PostgreSQL, but not both concurrently. However, it is platform dependent, being offered as a SaaS. Besides, Amazon Aurora can denote complex code maintenance, given that the implementation is effectively built inside the database code base. Therefore, although it allows the usage of both MySQL and PostgreSQL, the protocol is tightly merged to those databases code.

The development of a novel hybrid middleware replication protocol can rely heavily on a distributed log, increasing throughput without compromising availability and correctness.

Moreover, the Apache DistributedLog API is a core dependency of such novel algorithm, enabling replication with a high-level abstraction of data.

A hybrid replication middleware aims to be perceived as a replication protocol while drawing the benefits of database-specific implementations. The protocol enables replication through an autonomous protocol that is agnostic of the underlying database. It allows easy maintenance and configuration, since the purpose is to detach the replication from the data, yet maintaining the knowledge of write-sets and database transactions.

4

SQLWARE REPLICATION MIDDLEWARE

Distributed Logging mechanisms allow the growth of novel replication techniques and protocols, increasing the overall throughput and availability. Different replication protocols that rely on logging mechanisms manifest constraints and code complexity.

SQLware surpasses those shortcomings by relying on Apache DistributedLog and its hybrid middleware design, increasing throughput while maintaining high-availability, data consistency, and database independence.

The proposed replication system is implemented as a middleware layer, enabling a set of configurable criteria for tuning purposes. Moreover, in addition to the hybrid replication mechanism, the configurations provided allow to adjust the SQLware to act as a fully active or a fully passive replication mechanism.

SQLware provides two implementations, namely: a standard V-JDBC middleware, easy to use, fully configurable, and a high-performance direct API, for write-intensive tasks that do not require additional features.

4.1 MIDDLEWARE

The most established forms of database specific middleware interfaces are the *Open Database Connectivity (ODBC)* and *Java Database Connectivity (JDBC)*. These interfaces enable developers to write database agnostic code, and use that code with any underlying data source. Therefore, when changing the data source of the architecture, it is only necessary to reconfigure the system regarding the connectivity.

SQLware is developed in Java and it is based on the *Virtual Java Database Connectivity (V-JDBC)*. This tool is an interface that mimics the JDBC, offering all the core functionalities. Since the V-JDBC is effectively JDBC compliant, it can replace the default package of the underlying database, still allowing the usage of all the available functionalities. Moreover, it is possible to change the code of this tool, considering that it is an open-source code base.

V-JDBC is split into two components, the client, and the servlet that interacts directly with the database's default JDBC. Both systems communicate through the network, and stand in between the client and the underlying database. SQLware aims to offer the same

layer of abstraction and, consequently, it incorporates within the V-JDBC code base to offer a JDBC compliant interface to users.

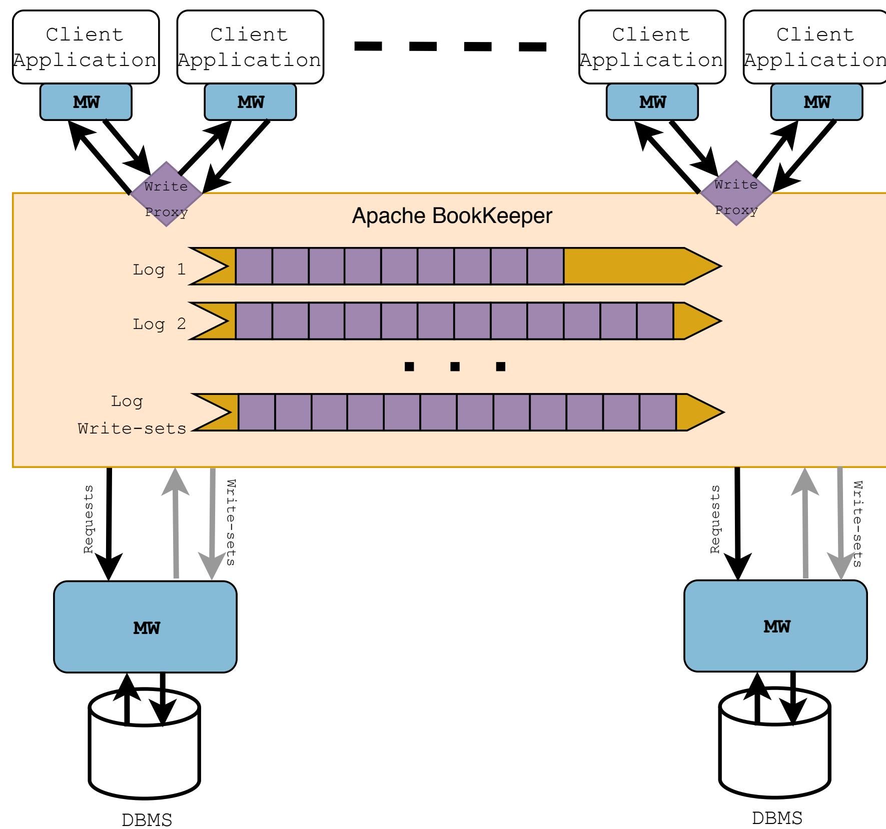


Figure 14: Hybrid middleware based replication: Comprehensive view.

The SQLware's V-JDBC prototype offers all core database functionalities, namely, the execution of generic queries. Therefore, SQLware can be seamlessly integrated into any system that uses JDBC. Moreover, SQLware V-JDBC's bird's eye view is depicted in Figure 14, offering a client interface and using a servlet to communicate with the underlying database default JDBC.

As illustrated in Figure 14, part of the middleware lives side-by-side with the database. This segment is the middleware's servlet that communicates directly with the database JDBC. Accordingly, if the database fails, the middleware service is assumed to fail as well, since the deploy of the servlet is accomplished alongside with the database. The middleware servlet is multi-threaded to endeavor high concurrency and solid performance, striving to avoid being the system bottleneck.

The middleware's portion of the system that lies on the client is provided as an interface that allows the seamless execution of generic queries. The middleware delegates the persistency to the write-proxies provided by the Apache DistributedLog. Moreover, the write-proxies guarantee a distributed write into durable storage, and the servlet is noti-

fied ensuring the query execution in the underlying relational database, writing back the write-sets to the durable and fault-tolerant storage.

Furthermore, the middleware assumes all the guarantees given by the Apache BookKeeper and DistributedLog API, described in Section 3.2.

4.2 SPECIFICATION

A multi-master scalable prototype is fully developed, improving performance and dependability. SQLware hybrid approach specification relies on a lazy approach, eventually executing the transactions, guaranteeing high-availability, high-throughput and data consistency.

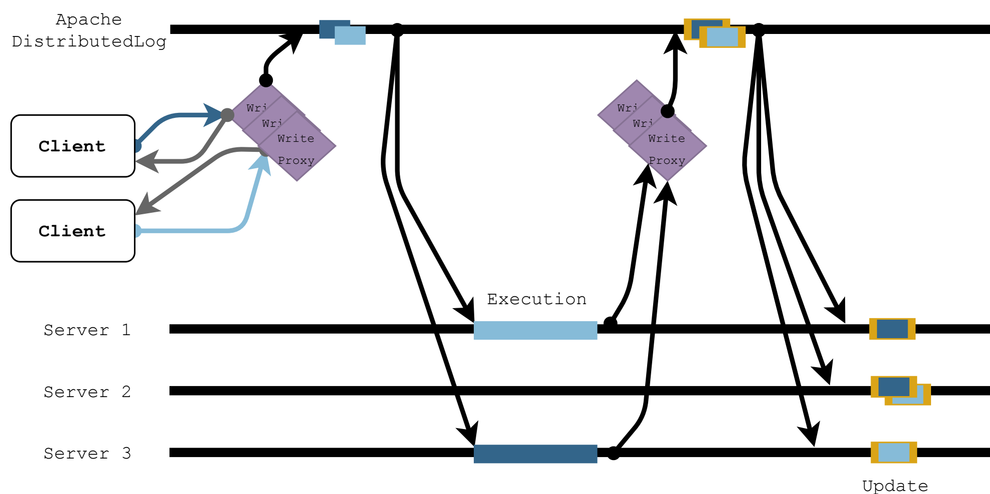


Figure 15: Hybrid middleware architecture flow.

Figure 15 depicts the multi-master specification. In this specification, all the clients must contact the Apache BookKeeper cluster through the DistributedLog API, using the deployed write-proxies, either by the V-JDBC middleware or the high-performance API provided. Clients contact a set of write-proxies (deploying as many as needed and, scattered throughout the globe as necessary), whose main features are the provision of fault-tolerance guarantees and manage fan-in writes from publishers.

Once the transactions are reliably persisted in durable storage, the write-proxies reply to the client. When the transaction is in the log stream, it is safe to reply to the client, since that transaction is available in a dependable storage, as it will be eventually executed by one of the database replicas.

As depicted in Figure 15, two clients can issue requests with different conflict classes, denoting the possibility that two replicas may execute those requests concurrently. Likewise, considering that both transactions have different conflict classes, their write-sets are conflict-free, and can be applied concurrently, increasing the overall system performance.

After the execution of a transaction, the write-sets are obtained using the database API and sent to another log stream. Other replicas apply those write-sets, maintaining a coherent state in the whole cluster.

In the event of a failure, the pending executions will be handled by other operating replicas, given that all machines in the system have access to all log streams and, consequently, no transaction is lost. Moreover, the designed system guarantees that the transaction is eventually executed as the DistributedLog API provides a read-ahead system, allowing to catch up with the log stream records. Such guarantees allow a multi-master system where all transactions are safely written on the log's durable storage, and executed by one of the many replicas in the system.

Besides a hybrid replication model, SQLware allows different configurations. Those configurations can shift the core specification to behave as a fully active technique or a fully passive one.

To accomplish an active replication technique with SQLware, all servlets are configured to open all available log streams, delegating the delivery and order of operations to the logging mechanism. The provision of strong consistency required by active replication requires a totally ordered set of transactions, provided by the logging mechanism and, to push records into the log stream, a log API is employed, either via the direct API or via V-JDBC.

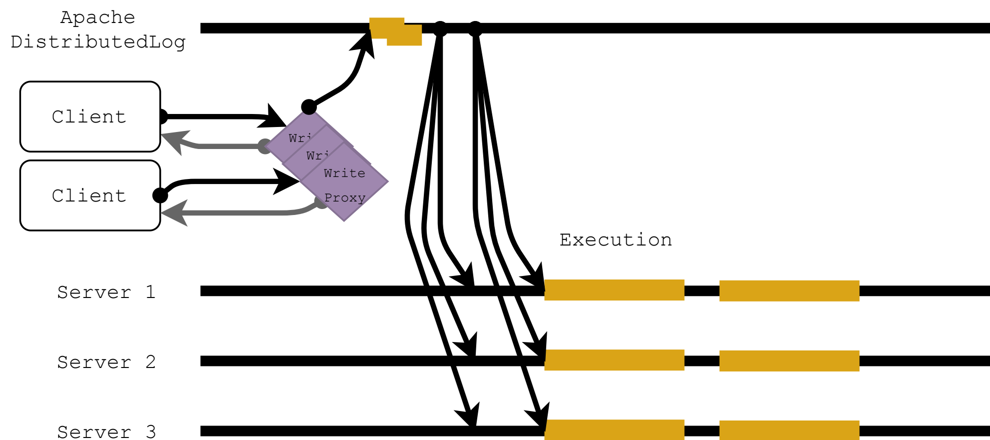


Figure 16: SQLware active replication.

Following the write of a new record in the log, all replicas receive a notification with that same record containing the transactions. Every transaction is assumed to commit as all have been ordered and execute sequentially. As of this implementation, delivery guarantees are not considered. Additionally, after completing the operation, the replica where the requests were delivered must reply to the client, as depicted in Figure 16.

The SQLware's passive replication technique deploys a single primary replica that opens all available logs. Therefore, only one replica processes requests, while backup replicas

apply updates received through the middleware, those updates are sent through another log stream that are opened by all backup replicas.

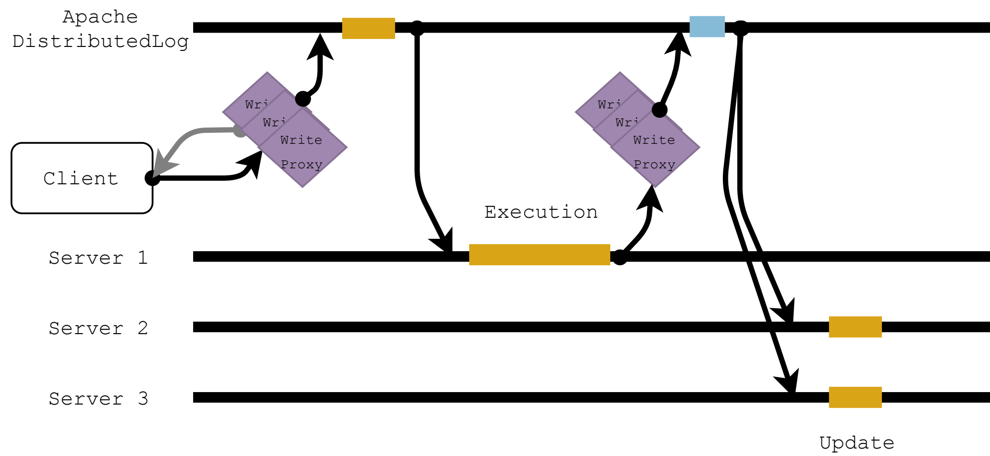


Figure 17: SQLware passive replication.

This mechanism requires the client to issue requests to the log through the middleware. Operations contained in the request are performed and, after commit, the backup replicas receive the deterministic updates. The middleware obtains the updates from the write-set service provided by the operational database. Write-sets are sent through the write-proxies to the log streams. When the records are persisted, all backup replicas receive a notification alongside with the record containing the updates to execute, as illustrated in Figure 17. Moreover, considering that this approach is merely a configurable scenario and, not the main technique, the primary replica failure is not considered since a leader election mechanism is not deployed.

4.2.1 *Backlog*

Backlog typically refers to an amount of work or processing, that is already issued, but not yet executed. Therefore, backlog and replica lag in this approach is expected, considering that, write-sets are sent through the network to other replicas. The concept of a backlog in this situation is associated with the number of transactions present on the log that are waiting for execution. However, it is also used to describe the number of write-sets waiting to be applied by other replicas. The first backlog is referred to as the requests backlog and, the second as write-sets backlog. Both backlogs are depicted in Figure 18, denoting all the critical points where requests and write-sets backlog can occur, specifically on the SQLware's hybrid configuration.

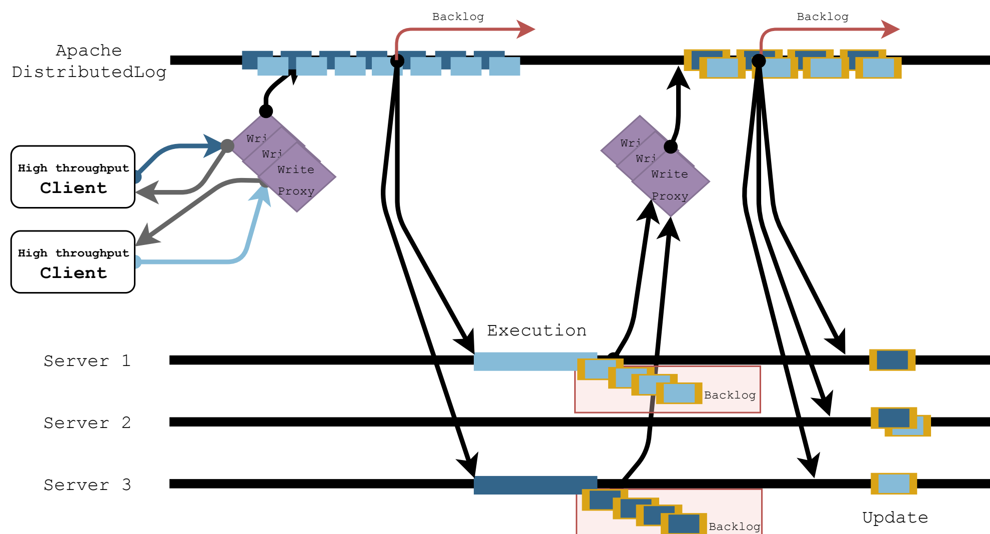


Figure 18: Backlog critical architectural points.

The backlog can get longer following a large batch of executions, as it normally happens when running benchmarks. The request backlog can get larger when the issued requests have higher throughput than the database. Considering that the database is concurrently processing requests, parsing write-sets and, applying write-sets, occasionally this type of backlog can happen, particularly when the underlying database is overwhelmed or resource limited.

As of write-sets, there are two ways from which the backlog can grow, either the database is limited by the WAL parsing performance or, similar to the requests backlog, the database is limited performance-wise. The first write-set backlog can occur by having too many connections executing concurrently or, if the write-sets are too big, occupy too many resources applying changes. Therefore, enabling the write-sets to grow on the Log. On the other hand, the second type of write-set backlog can happen if the database API considered to obtain write-sets is too slow. Likewise, the backlog can become larger even before being sent to the log streams, denoting that the write-sets backlog is within the database, and not in the log streams. Therefore, in this case, the database becomes a bottleneck since it is too slow performing the parsing, and returning it to the middleware. Accordingly, a configuration file is available so that users can configure the number of lines read from the database write-sets, trying to control or avoid this bottleneck.

4.2.2 *Assumptions*

The implementation of the middleware is supported by two necessary assumptions, allowing a multi-master approach and enabling a core functionality of the protocol, the write-set broadcast.

- **Data partitions**

  The first assumption is that the dataset can be partitioned, denoting that conflict classes can be split and assigned to different replicas. Consequently, transactions executed in different replicas are performed concurrently, since it is possible to acquire locks for known conflict classes. The partitioning of the dataset is known as horizontal partitioning and, it is a common practice in highly distributed databases.

  It is important to notice that, if such assumption is not possible, this solution can still be applied since it is possible to configure the middleware to act as a fully active or, a fully passive protocol. Moreover, by configuring the SQLware to serve such techniques, there are performance bottlenecks considering that there is no load balancing. Although such a shift might slow down the solution, it is still feasible since the prototype induces a low impact on performance.

- **Write-sets retrieval service**

  The second assumption is necessary to enable the middleware to send and apply write-sets. It is expected that the underlying database provides one of two services. Either it provides one service to obtain write-sets in the form of deterministic queries, allowing to apply them in remote sites effortlessly or, one bundled service to obtain write-sets, in whatever shape or form, alongside with another to interpret and apply such updates.

4.2.3 *Horizontal partitioning*

Horizontal partitioning allows the distribution of data into subsets, allocating those subsets to different nodes, areas, devices and files. Horizontal partitioning in distributed databases has been used for years and, the data partitioning feasibility can be defined probabilistically [5]. Acknowledging that it is mathematically possible to determine the data partition, the developed solution assumes a dataset distribution, allowing to balance the load through different replicas, and increasing the system performance overall.

Despite the theoretical base of the horizontal partitioning, the implementation in this solution relies on sharding only to manage load balancing. Moreover, the system maintains all replicas with the complete database, as detailed in the architecture. Accordingly, the partitioning solely occurs when issuing, and while executing requests. Following the

execution, databases respectively send the updates to the other machines, maintaining a coherent state in the whole cluster.

4.2.4 *Underlying Database Management System*

The middleware is built around a standard interface that enables clients and operational databases to connect. Therefore, a set of available configurations allow the user to change the underlying database. Moreover, the integration of the middleware with other databases depends on a interface, enabling the database development community to apply their solutions. The implementation depends on two main classes, a *DBEngine* and a *Log*. Both interfaces need to be developed for each database, since there is no universal solution to obtain write-sets from databases, acting as a driver.

Virtually every database has a WAL decoding API, allowing to obtain write-sets. Considering that those APIs change from one database to another, to enable the implementation and development of this solution, the PostgreSQL database [19] is chosen for its flexibility, and write-ahead log decoding features. Although those decoding features exist in different databases, PostgreSQL community had already developed plug-ins and open-source tools to enable different encodings for write-sets, empowering an extensive selection for write-sets execution and serializing characteristics.

For the implementation of this middleware, the native logical replication is replaced by a plug-in [9], manipulating the logical replication mechanism to achieve an output mode. Therefore, instead of having subscribers connected to PostgreSQL replication slots, these slots are manipulated by the plug-in to return the WAL deterministic queries in a result-set. Once configured, the plug-in enables the middleware to explicitly request write-sets, enabling the broadcast of those updates to backups, hence not requiring any native replication.

4.2.5 *Configuration parameters*

The most relevant configuration regarding this solution is the underlying database configuration. As a middleware solution, the underlying data source can be freely configurable, from the Driver used, to the number of connections available in the provided connection pool. That connection pool is used by write-set readers to apply write-sets of other replicas. Therefore, the more connections, the faster the treatment of those updates.

Configuration	Default	Description
db.driver	org.postgresql.Driver	Database JDBC driver
db.uri	jdbc:postgresql://localhost:5432/tpcc	Database location
db.user	postgres	Database username
db.password	123456789	Database password
db.pool	8	Database connection pool
db.engine.class	PostgreSQLEngine	Path to class of engine
db.log.class	PostgreSQLLog	Path to class of log

Table 1: SQLware underlying database configuration parameters.

Table 1, depicts the configurable parameters concerning the underlying database connection and log class. It is possible to set up every component of the connection. Both the parameters *db.engine.class* and *db.log.class* are used to load the implementation of those classes. The provided classes, log, and engine are loaded using the java reflection feature with dynamic class loading [14]. By default, the PostgreSQL classes are already available, since it is the database adopted to prototype the solution.

Configuration	Default	Description
dlog.reader.numThreads	8	Number of thread to apply updates
db.writeset.sharding	false	Whether it is possible to shard data
db.writeset.nReaders	1	Number of database write-set readers
db.writeset.nTransactions	1000	Number changes obtained from the database log
db.writeset.delay	0	Delay between write-set reads
db.writeset.timeout	30	Time-out when writing write-sets to Log
db.writeset.bulk	false	Whether to bulk writes for Log

Table 2: SQLware underlying database write-set service parameters

Table 2 depicts the parameters with the highest impact on system's performance, either increasing it or restricting it. One of these parameters (*dlog.reader.numThreads*) allow setting the number of threads that apply write-sets from other machines. However, thread concurrency is ultimately bound by the number of connections available in the pool. It is also possible to configure the partition of write-sets, and how many concurrent threads are used to obtain those write-sets, ultimately writing them to the log.

Another significant parameter is the number of lines retrieved from the write-set socket (*db.writeset.nTransactions*). This parameter can affect performance since by reading too many lines, a large result-set will have to be processed by the database, delaying the response. If a small number of lines is configured, there will be saturation in the number of writes in the log. In conjunction, the write delay parameter (*db.writeset.delay*) can also be set, allowing a user to configure what best suits the application.

Configuration	Default	Description
dlog.writeproxy.finagle	zk!**ip**:**port**!/messaging /hybrid/.write_proxy	Write-Proxy Finagle
dlog.uri	distributedlog://**ip**:**port** /messaging/hybrid	DistributedLog URI
dlog.writeset.log	wal	Log to write write-sets
dlog.readset.logs	wal	Logs to read write-sets
dlog.logs	log-1, log-2, log-3	Requests logs to handle

Table 3: SQLware DistributedLog API configuration.

Table 3 presents all the configurations required for the DistributedLog API setup. The required parameters are the finagle *Uniform Resource Identifier (URI)*, being a single IP or an Apache ZooKeeper sustained membership and, the DistributedLog bound URI. In these configurations the Apache BookKeeper cluster is transparent. Accordingly, solely the Apache ZooKeeper IP and port, the DistributedLog specific URI and, the write-proxies finagle definition are configurable.

5

SYSTEM ANALYSIS AND RESULTS

SQLware is evaluated with the industry standard TPC-C benchmark [24], specifically, the Escada TPC-C [12]. The goal is to identify the bottlenecks of the protocol while achieving the highest performance possible without compromising availability neither correctness. Consequently, the high-performance API will be used, bypassing the V-JDBC interface and avoiding the extra latency. By using the direct API, the Apache BookKeeper cluster is reached directly, abstracting all JDBC functionalities.

5.1 EXPERIMENTAL SETTING

The TPC-C specification models a real-world scenario where a company, comprised of several warehouses and districts, processes orders placed by clients. The workload is defined over 9 tables operated by a transaction mix comprised of five different transactions, namely: New Order, Payment, Order-Status, Delivery and Stock-Level. Each transaction is composed of several read and update operations, where 92% are update operations, which characterizes this as a write heavy workload. The benchmark is divided into a load and an execution stage. During the first stage, the database tables are populated and, during the second stage, the transaction mix is executed over that dataset. TPC-C defines how these tables are populated and also defines their size and scaling requirements, which is indexed to the number of configured warehouses in the system. The outcome of this benchmark is a metric defined as the maximum qualified throughput of the system, *tpmC*, or the number of New Order transactions per minute.

This benchmark scales with the number of warehouses. As long as the number of warehouses increases, the total number of clients also increases, as it is bound by the *Warehouse (WH)* configuration.

The evaluation was deployed over two setups. The first setup comprises a single replica machine that allowed to execute the first group of micro-benchmarks to verify the architecture correctness and determine the comparisons to be performed. The second setup is based on a two replica composition, on a HPC environment. In both setups, the database considered was deployed in Docker containers [11], enabling a fast development environ-

ment by establishing simple and straightforward connections between databases and the developed middleware. Each container holds a single database instance, and the number of deployed containers varies according to the setup.

The first batch of benchmarks are performed in the first setup, with the following hardware:

- Intel® Xeon® CPU E5-2670 v3 @ 2.30GHz with 48 cores, 99 GB of RAM and a 6 TB shared volume storage with HDDs.

On this setup two database containers are deployed. This setup is adopted considering the micro-benchmarks to be run. Taking into account that these benchmarks are executed to understand the feasibility of the prototype and, set the initial configurations to be used, considered as baseline.

Both databases are configured to write into a network storage based in HDD's and, both share all the machine's resources, namely, CPU, RAM and, bandwidth to the storage. Moreover, a three bookie ensemble is simulated with a sandbox, being configured to write to the machine's local disk, also sharing the machine's resources. However, all distributed writes are redirected to the same disk, meaning that the bottleneck of this machine is expected to be reached swiftly as the number of warehouses increases.

In the second setup, tests are executed in an HPC structure with two physical replicas:

- Intel® Xeon® Platinum 8153 CPU @ 2.00GHz with 256 CPUS, 3TB of RAM and an array of 24 HDD disks.

- Intel® Xeon® Platinum 8158 CPU @ 3.00GHz with 192 CPUS, 4TB of RAM and an array of 24 SSD disks.

Both the machines are connected through a 10 GB network, maintaining a low latency between them.

In this HPC environment a full BookKeeper cluster is set up, in contrast to the single replica that used the sandbox provided. In this setup a fully deployed cluster is adopted, considering that enough local disks are available and, as an HPC environment, it is possible to configure a fully distributed deployment. Therefore, a three bookie ensemble, having two bookies in one machine and one bookie on the remainder machine are deployed on this environment. All the bookies are isolated by disk, alongside with their journal logs, exploiting the I/O parallelism [22]. Moreover, both replicas have a database instance that also maintains their data in an isolated disk to maximize the parallelism and to adequately simulate a fully distributed environment.

5.2 CONFIGURATIONS

While executing the first and second set of benchmarks, different configurations on the middleware had to be tested, considering that both systems are distinct. When executing with different configurations, it is possible to acknowledge that the parameters defined induced an impact on performance. Considering that it is infeasible to experiment every configuration value, achieving the setup that maximizes the performance, a balanced configuration is established after numerous attempts. The values chosen allowed a steady tpmC in comparison with similar solutions, including the native replication of the underlying database.

An important aspect to discuss is the sharding considered in Section 4.2.2. As assumed, the implemented middleware relies on horizontal partitioning to guarantee the multi-master architecture and high performance. As such, SQLware leverages the TPC-C's dataset, allowing a simple and effective data partition. The sharding of the dataset is performed per warehouse. By partitioning the dataset per warehouse, it is possible to assign each warehouse to a replica allowing a balanced workload between replicas and, the use of multi-master, as the conflict classes are distinguished per each warehouse.

Changing the configuration parameters can affect not only the tpmC, but also the backlog of the protocol. Some specific adjustments on configurations can affect both metrics, increasing the tpmC, and reducing the backlog.

In the middleware configuration, a finite amount of specific points needed to be adjusted to run the benchmarks, in order to reach the best possible throughput, namely:

1. **Warehouses per log**

   This parameter has a significant impact on performance overall, considering that writing a large number of warehouses on a single log stream can hurt performance. However, writing a small number of warehouses also harms performance, because there is the need to handle a wide group of log streams.

2. **Write-set writers**

   This configuration can be tuned to improve throughput and help reduce the replica lag. Similar to the number of WHs per log, this configuration has the potential to affect performance. The database write-sets can be rather large and, although deploying a minority of writers make them write large packages ever so often, when deploying a large number of writers, it will make them write a comprehensive quantity of small packages, also saturating the log.

3. **Number changes obtained from the database log**

   If this parameter is set, reading write-sets will halt when the number of rows produced exceeds the specified value, yet, the exact amount of rows can be larger. The

limit defined is only verified per transaction, therefore transactions will never be compromised.

4. **Writing delay**

   This delay will define the time waited between write-set writings on the log. Once written a full write-set, the mechanism will wait the specified time to retrieve more write-sets. This configuration can be useful for small write-sets when there is a need to get a larger one to reduce the latency of writing on the log, in that case, the mechanism can wait a few seconds before getting more write-sets, obtaining a larger write-set rather than a small one, avoiding a constant pooling to the database.

5. **Batched writes**

   This option describes if the write-set records are written in batch. This parameter enables the middleware to bulk write multiple records using a primitive function provided in the DistributedLog API. Writing aggregated packages can affect latency and throughput. Therefore, this configuration depends on the application needs. If this option is set to off, the records will be written using the normal function, individually.

These are not the only configurable points since components also maintain performance configuration themselves, from the underlying database to the write-proxies. Those configurations are not outlined in this section since, in what regards the DistributedLog, the production default configuration is used.

The next section will describe the analysis of the benchmark results using the following configuration parameters:

Configuration	**50w**	**100w**	**200w**	**400w**
WHs p / log	25	50	100	200
db.pool	16	16	16	16
dlog.reader.numThreads	32	32	32	32
db.writeset.sharding	*true*	*true*	*true*	*true*
db.writeset.nReaders	2	2	2	2
db.writeset.nTransactions	100.000	100.000	100.000	100.000

Table 4: SQLware configuration values for the single machine setup.

Configuration	100w	1000w	2000w	3000w
WHs p / log	10	50	100	150
db.pool	30	30	30	30
dlog.reader.numThreads	128	128	128	128
db.writeset.sharding	*true*	*true*	*true*	*true*
db.writeset.nReaders	1	4	4	4
db.writeset.nTransactions	0	100.000	100.000	100.000

Table 5: SQLware configuration values for the HPC setup.

The configurations not stated remained in the default values. Other parameters like the *logs* are omitted since it is a verbose list of the logs opened. Therefore, only the relevant values are detailed.

5.3 RESULTS

This section discusses the results achieved by running TPC-C benchmark. The analysis performed is focused on the benchmark results and, on the comparison of similar techniques, discussing the improvements and bottleneck associated with this solution.

5.3.1 *Single Replica environment*

The first micro-benchmarks are run in a single replica environment, and a diverse number of benchmarks performed on this machine granted extensive knowledge about the correct comparison between the techniques. When adjusting different configurations, replication guarantees and dependability became the crucial aspects to consider. Therefore, from *fsync=off* to *synchronous_commit=on*, a wide range of database configurations are benchmarked, narrowing down the alternatives.

Considering the wide choice of configurations tested, the most similar ones are the middleware solution, the PostgreSQL asynchronous replication, and the PostgreSQL synchronous replication with the configuration *synchronous_commit=on*.

Warehouses	Abort rate (%)	Avg latency (ms)	tpmC
50w	4,3	503,60	1062,59
100w	2,5	633,62	856,33
200w	1,0	836,15	643,78
400w	1,0	1183,80	433,92

Table 6: PostgreSQL synchronous replication benchmark values.

The values displayed in Table 6 are the PostgreSQL synchronous replication benchmark results. It is possible to observe the correlations between throughput, latency and abort rate. The throughput of this technique is continuously decreased as the number of warehouses increases, making the total number of terminals grow. Therefore, the abort rate is smaller and higher latency is experienced, as expected.

Warehouses	Abort rate (%)	Avg latency (ms)	tpmC
50w	3,9	252,42	2104,32
100w	2,0	264,65	2019,66
200w	1,0	373,48	1433,21
400w	0,6	371,66	1463,29

Table 7: PostgreSQL asynchronous replication benchmark values.

In comparison with the previous protocol, the native asynchronous replication has noticeable improvements in performance as expressed in Table 7. The biggest drawback regarding this technique is, in fact, the dependability guarantees. As detailed, this technique manifests a 2 times increase in the smallest benchmark and nearly a 4 times increase in the largest, providing this technique a clear advance performance-wise.

Warehouses	Abort rate (%)	Avg latency (ms)	tpmC
50w	0	54,88	5915,08
100w	0	58,80	9067,37
200w	0	58,40	9300,83
400w	0	59,35	9172,23

Table 8: Middleware replication benchmark values.

SQLware results are demonstrated in Table 8, and as expected, the abort rate is null considering that the TPC-C client uses the high-performance API, interacting directly with the Apache BookKeeper cluster, bypassing the V-JDBC latency. Besides, since the requests are sharded by warehouse before being written on the log streams, individual shards are executed sequentially on the database replicas.

The overall throughput is increased, denoting that the SQLware is 6 times faster than the underlying database asynchronous replication, and 20 times faster than the synchronous one. These results are obtained by configuring two logs and, when executing the benchmark, the workload is balanced between both logs.

The relevant results obtained by benchmarking the system, besides being represented in the previous tables, are also represented in a two-axis graph. Figure 19 represents the tpmC value of each protocol as the amount of warehouses increases, denoting the overall throughput of the techniques. The fluctuation of values is undoubtedly large. It is possible

to observe both native replication techniques halting behind the hybrid middleware by a considerable disparity.

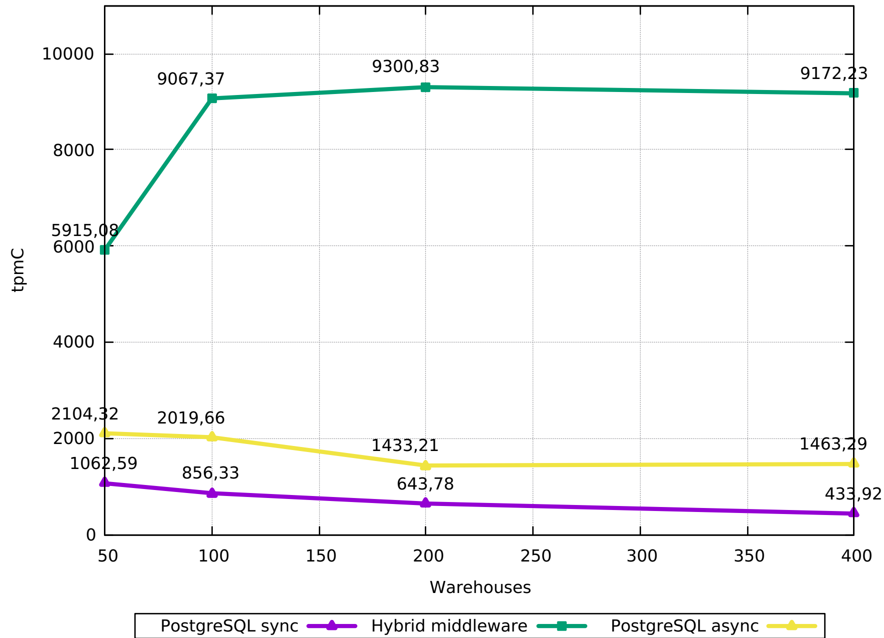


Figure 19: Single machine benchmarks comparison.

The biggest bottleneck on this machine lied on the durable storage. Considering that, the available storage unit is sustained by hard disk drives and, our solution relies heavily on durable storage, the results will reflect the HDD write speeds. Moreover, when loading a larger dataset (i.e., the 400 WH dataset), the disk cache occupied all the available RAM. Consequently, just 300 MB of free memory remained free, requiring the machine to swap the RAM to an HDD slowing down the entire setup and, heavily increasing the amount of backlog.

Although the backlog obtained by running the benchmark with the 400 WH is significant, it is overlooked considering that it represents a side-effect produced by the shortage of RAM. With this environment, the backlog bottleneck is unmistakably the 400 WH dataset.

Furthermore, with an HPC environment and two distinct replicas, the backlog is small, even when increasing the dataset to approximately 8 times the current size. Therefore, given the results obtained in this environment and, after understanding the order of magnitude regarding the values achieved, there is a shift towards an HPC environment.

5.3.2 *High Performance Computing environment*

When shifting the setup to the HPC environment, the benchmarks executed are limited to the PostgreSQL native synchronous replication, and the hybrid middleware replication, considering that those techniques are the closest and the most relevant to be compared.

Although the synchronous replication of PostgreSQL waits until the slave replica writes the transaction to disk, in the developed solution this is not a requirement. Within the developed middleware, transactions are issued to the log and, consequently replicated in the three bookies, guaranteeing a distributed write. Being the transaction written in a distributed manner, if a first replica fails, the second one can easily replay transactions from the log, guaranteeing that transactions are eventually fulfilled. Such guarantees are not satisfied in the asynchronous replication of PostgreSQL. If a transaction is executed in the master replica and the client receives a response, in the event of failure after executing the transactions, there is no guarantee that the slave replica has received that transaction or, even if it will ever get it.

Consequently, in this section, only the PostgreSQL synchronous replication and, the middleware solution will be compared. These benchmarks will allow comparing two techniques that guarantee a replicated write (i.e., a guaranteed write in more than one replica).

Although the order of magnitude when shifting to this environment is increased tenfold, there is a constraint regarding the scalability of the benchmark dataset, since the disks on both machines have a maximum capacity of 512GB. Therefore, the durable storage does not support a dataset larger than than 3000 WHs, limiting the comparison. Consequently, there will be increases of 1000 WHs at a time, starting from the 100 WHs dataset.

Warehouses	Abort rate (%)	Avg latency (ms)	tpmC
100W	2,6	320,95	1.668,46
1000W	0,62	710,84	698,06
2000W	0,62	741,95	679,82
3000W	0,54	761,35	670,38

Table 9: PostgreSQL synchronous replication benchmark values on HPC.

Table 9 depicts the values obtained by the synchronous replication of PostgreSQL. In terms of comparison, it is possible to understand that, contrary to the single machine setup, this environment has 2 times more throughput on the 100 WHs benchmark and, it maintained the throughput value as the scaling factor increased, sustaining it in the 600 tpmC.

Warehouses	Abort rate (%)	Avg latency (ms)	tpmC
100W	0	4,43	11.870,66
1000W	0	4,92	102.748,15
2000W	0	4,98	103.861,43
3000W	0	5	104.006,84

Table 10: Hybrid replication benchmark values on HPC.

As depicted in Table 10, SQLware in comparison with the synchronous version of PostgreSQL replication, details a difference of 150 times more throughput. Consequently, the average latency is also decreased by 150 times, showing the robustness of the designed protocol.

Moreover, it is possible to establish a comparison with the single machine environment, showing an increase of 11 times times more throughput and 10 times less latency. Even though the BookKeeper cluster and PostgreSQL replicas are sharing bandwidth, processing power and memory, the throughput values remained unwaveringly at the 100.000 tpmC.

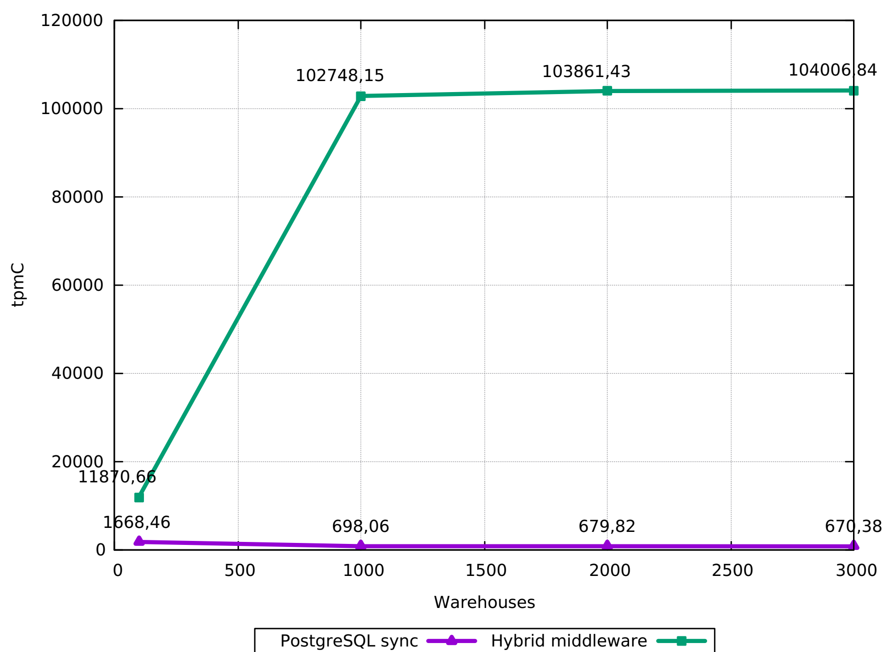


Figure 20: HPC benchmarks comparison.

The values detailed on the previous tables are depicted in Figure 20, allowing to represent the values distinction. It is possible to verify the middleware scaling and stable throughput when achieving the 1000 WHs.

Alongside with the throughput increase, there is also a registered backlog increase. In the current environment, the backlog is fluctuating between 5 seconds to 30 seconds, depending

on the configuration. However, the calculated average replica lag is 6 seconds, determining that on average, replicas need 6 seconds to stay consistent with each other. Since a SQLware replica is concurrently executing requests and applying write-sets from other replicas, there is replica lag and request latency. On average the latency of the requests never surpassed 5 seconds on the slowest replica. The requests latency and replica lag can be heavily reduced by introducing more database replicas.

The mentioned replica increase is not deployed since the environment only operated two physical machines. Although this deployment is only on a two replica cluster, as it is an HPC environment and, the overall system capacity is not fully occupied, given the I/O parallelism, amount of CPU power and RAM, the Apache BookKeeper cluster has enough processing power and throughput to operate as a dedicated environment. Moreover, the databases also performed as expected, delivering performance without affecting the overall throughput.

In both scenarios, after the backlog execution, both databases rested on a consistent state with each other, fulfilling both high-performance, high-availability, and data consistency goals of this dissertation.

5.4 DISCUSSION

As expected, the main bottleneck of the whole system is durable storage. Every transaction is always persisted in durable storage throughout the Apache BookKeeper cluster. Consequently, with an *ack* quorum of three bookies in a three replica ensemble, the slowest the disks the worst the tpmC.

Also, the system RAM is also a clear bottleneck in this approach. If the system RAM is not enough to hold the entire data-set, a large backlog can occur, decreasing the overall system performance.

In this PostgreSQL environment, an SQL query based plug-in is used. This plug-in allows translating the database write-sets into executable queries. One of the parameters possible to define is the number of rows obtained by the database. If this number is fixed excessively, the database will throttle in the write-set parsing, since it is synchronous and sustained by a single-threaded operation. However, if a small amount of lines is defined, a large number of writes per second can occur in the log, overwhelming the log stream.

The previously mentioned bottlenecks can be surpassed with some techniques, some of these improvements are addressed and detailed in the following section, while others were already detailed in the previous configuration section.

Since it is possible to configure the middleware to suit the needs of each application, there is room for improvements. Although some configurations can improve replica lag and boost the number of transactions per minute, there are others that can cause a performance decay.

One of the improvements implemented in the coded PostgreSQL interface, is the usage of sharding while obtaining write-sets. Therefore it is possible to issue multiple writes, making this configuration one of the most important adjustments that can either improve or deteriorate performance, alongside with the number of write-set transactions obtained by the database API.

Furthermore, by issuing a large number of write-set writers (since write-sets have typically, a considerable size), writing extensive updates will decay the number of transactions for the clients. This effect is rather common, as the cluster is shared by both the requests and write-sets. Therefore, it is possible to overwhelm the cluster with large write-sets since the same bookies and disks are being used.

Another possibility to improve the overall system performance is to increase the number of bookies since Apache BookKeeper implements a policy that improves reads and writes, by distributing the various log segment across different bookies, as stated in the replication Section 3.2.4. Accordingly, if the bookies are increased, also the performance is expected to do so and, the following hypothesis is clearly understandable since the scalability is horizontal, and the system load is balanced throughout the whole cluster.

Moreover, if the amount of databases is increased horizontally, the write-set API bottleneck is also reduced, since each database will handle a smaller sub-set of transactions, reducing the overall size of the write-sets per database.

6

CONCLUSIONS AND FUTURE WORK

6.1 CONCLUSION

This thesis is born from the observation that so far, replication protocols such as the active, passive and even hybrid branches of the former protocols have important limitations. Moreover, Distributed Logging can become an effective source for throughput improvement as it has been seen in previous work as in Amazon's Aurora [25]. Therefore, this thesis strived to develop a middleware solution, providing seamless replication with high-availability, fault-tolerance guarantees, and high-throughput by considering a hybrid replication protocol with Distributed Logging as foundation. As a middleware, the proposed solution aimed to be database agnostic and easily maintainable.

SQLware is based on a three-layer solution, the client middleware, the Apache BookKeeper log streams and, finally, the servlet middleware in conjunction with the underlying database. The prototype is developed with PostgreSQL as the underlying database, leaving the interfaces open to other data sources. SQLware aims to be used as a JDBC driver, offering a seamless usage of the replication protocol. The most important layer is effectively the Apache BookKeeper, offering a highly distributed log store, fault-tolerance and huge write performance. The lowest tier layer lies on the servlet middleware, aiming to live side-by-side with the database server, building a bridge between the underlying database and the rest of the architecture.

The main features offered by SQLware are, a high-availability middleware, able to interact with any underlying data source, offering high-throughput and low latency. Fault-tolerance guarantees are provided to the end-user, allowing to build a highly scalable system, tolerant to system failures and network partitions.

Through analysis and benchmarking using the industry standard TPC-C benchmarking tool, the prototype confirmed an enormous improvement in comparison to the baseline. The middleware is compared with the native replication of the underlying data source, the PostgreSQL native replication.

Compiling the results and values, using the TPC-C, the middleware provided up to 150 times more throughput than the native replication offered by the underlying database, which not only corroborates the thesis, but highlights the positive impact on using distributed logging as part of the replication mechanism. Moreover, in addition to the performance increase, the middleware does not compromise the availability, and the overall tolerance to system and network failures.

6.2 FUTURE WORK

There are two essential aspects to guarantee a mature development, and a production-ready solution. First and foremost, the recovery mechanism, allowing the underlying database to recover from failures. Secondly, a dynamic configuration system implemented in the middleware code base.

A recovery mechanism for SQLware is necessary, as it is for any replicated database, either to allow a database to recover from an inconsistent state or to allow new replicas to obtain a coherent copy. Since the log streams used are totally ordered, they can be read using the DistributedLog Sequence Number (DLSN). Therefore, it is possible for one given replica to continue the work of others when a failure occurs (i.e., fail-over) and, to a new replica to apply every write-set written from the beginning. Although a straightforward description is given in this excerpt, more sophistication to the recovery protocol is required (e.g., full copies and snapshots). Therefore, efforts to build such recovery mechanism are concurrent with the SQLware development, as both projects are small parts of a whole, ultimately achieving a full integration.

As discussed, different configurations affected both throughput and backlog in different ways. Therefore, some trade-offs need to be made and, those decisions depend on each application and its purposes. Consequently, a dynamic configuration system would allow configuring numerous replicas without the need to restart the servlet, allowing to take into account the system load and replica lag. Such a system could be based on Apache ZooKeeper, given that it is already deployed in the cluster and is identified as core dependency of the architecture. Moreover, a dynamic configuration system is desired since not every single user has advanced know-how of its own application. Therefore, this system can allow an intelligent configuration based on system parameters and, still allowing advanced users to be able to change those parameters to suit their needs.

Both points addressed will allow SQLware to mature into a production-ready environment, providing an overall vision of future prospects to implement such protocol.

BIBLIOGRAPHY

[1] Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases. In *European Conference on Parallel Processing*, pages 496–503. Springer, 1997.

[2] Apache BookKeeper - A scalable, fault-tolerant, and low-latency storage service optimized for real-time workloads, 2018. URL https://bookkeeper.apache.org/.

[3] Apache ZooKeeper, 2018. URL https://zookeeper.apache.org/.

[4] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[5] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the 1982 ACM SIGMOD international conference on Management of data*, pages 128–136. ACM, 1982.

[6] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[7] Alfrânio Correia, José Pereira, and Rui Oliveira. Akara: A flexible clustering protocol for demanding transactional workloads. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 691–708. Springer, 2008.

[8] Flaviu Cristian, Houtan Aghili, Raymond Strong, and Danny Dolev. *Atomic broadcast: From simple message diffusion to Byzantine agreement*. Citeseer, 1986.

[9] decoder_raw, Output plugin for logical replication, 2018. URL https://github.com/hugomiguelabreu/pg_plugins.

[10] Xavier Defago, Andre Schiper, and Nicole Sergent. Semi-passive replication. In *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pages 43–50. IEEE, 1998.

[11] docker, 2018. URL https://www.docker.com/.

[12] Escada TPC-C is an easy to use JDBC benchmark that closely resembles the TPC-C standard for OLTP. DB's supported include PostgreSQL, MySQL, and Derby., 2018. URL https://github.com/rmpvilaca/EscadaTPC-C.

[13] European Central Bank, 2017. URL https://www.ecb.europa.eu/press/pr/stats/paysec/html/ecb.pis2017.en.html.

[14] Ira R Forman, Nate Forman, and John Vlissides Ibm. Java reflection in action. 2004.

[15] Bettina Kemme and Gustavo Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB*, pages 134–143, 2000.

[16] Tadeusz Kobus, Maciej Kokocinski, and Pawel T Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 286–296. IEEE, 2013.

[17] Fernando Pedone, Rachid Guerraoui, and André Schiper. Exploiting atomic broadcast in replicated databases. In *European Conference on Parallel Processing*, pages 513–520. Springer, 1998.

[18] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[19] PostgreSQL, 2018. URL https://www.postgresql.org/.

[20] David Powell, Marc Chérèque, and David Drackley. Fault-tolerance in delta-4. *ACM SIGOPS Operating Systems Review*, 25(2):122–125, 1991.

[21] Reddit statistics, 2018. URL https://pushshift.io/.

[22] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal—The International Journal on Very Large Data Bases*, 7(1):48–66, 1998.

[23] Fred B Schneider. Replication management using the state-machine approach. *Distributed systems*, 2:169–198, 1993.

[24] TPC-C Benchmark - Standard Specification, February 2010. URL http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[25] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052. ACM, 2017.

[26] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *Distributed*

Computing Systems, 2000. Proceedings. 20th International Conference on, pages 464–474. IEEE, 2000.

[27] Wordpress, 2018. URL https://wordpress.com/activity/.

INESCTEC
TECNOLOGIA E CIÊNCIA

HASLab
HIGH-ASSURANCE
SOFTWARE LABORATORY

CloudDBAppliance