Helder Ricardo Laximi Martins

Licenciado em Engenharia Informática

Distributed Replicated Macro-Components

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : João Lourenço, Prof. Auxiliar,
Universidade Nova de Lisboa

Co-orientador : Nuno Preguiça, Prof. Auxiliar,
Universidade Nova de Lisboa

Júri:

Presidente: Prof. António Maria Lobo César Alcarão Ravara
Universidade Nova de Lisboa

Arguente: Prof. Hugo Alexandre Tavares Miranda
Universidade de Lisboa

Vogal: Prof. Nuno Manuel Ribeiro Preguiça
Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Setembro, 2013

Distributed Replicated Macro-Components

Aos meus pais, pela oportunidade proporcionada.

Acknowledgements

I am thankful to my advisers, João Lourenço and Nuno Preguiça, not only for accepting me as their student and introducing me to the world of research, but also for their guidance, advices, devotion and for being patient with me during the elaboration of the dissertation. I would also like to extend my gratitude to João Soares for his support when I needed it. I am grateful for participating in the RepComp project (PTDC/EIA-EIA/108963/2008) that partially funded this work.

I am grateful to Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, for granting me with four scholarships during the M.Sc. course, which allowed me to carry on.

I am also thankful to my co-workers and friends who frequented the room of the Arquitectura de Sistemas Computacionais group, for all the moments of work, fun and despair that we shared. None of this would be possible without their presence.

To my parents, I am immensely grateful for their support, guidance and understanding in my bad humor days and specially for providing me with this opportunity. To Joana, for her endless love that fills my heart and helped me through all of my bad times giving me encouragement to go on.

Last but not least, I also wish to thank all of my family for their support. In particular, I want to thank my friends João Silva, Diogo Sousa, João Martins, Andy Gonçalves, Lara Luís, Laura Oliveira, Nuno Pimenta, Gabriel Marcondes and the rest of my colleagues for also sharing all of my good and bad moments and making my day better when I needed it the most.

Abstract

In recent years, several approaches have been proposed for improving application performance on multi-core machines. However, exploring the power of multi-core processors remains complex for most programmers. A *Macro-component* is an abstraction that tries to tackle this problem by allowing to explore the power of multi-core machines without requiring changes in the programs. A Macro-component encapsulates several diverse implementations of the same specification. This allows to take the best performance of all operations and/or distribute load among replicas, while keeping contention and synchronization overhead to the minimum.

In real-world applications, relying on only one server to provide a service leads to limited fault-tolerance and scalability. To address this problem, it is common to replicate services in multiple machines. This work addresses the problem os supporting such replication solution, while exploring the power of multi-core machines.

To this end, we propose to support the replication of Macro-components in a cluster of machines. In this dissertation we present the design of a *middleware* solution for achieving such goal. Using the implemented replication *middleware* we have successfully deployed a replicated Macro-component of in-memory databases which are known to have scalability problems in multi-core machines. The proposed solution combines multi-master replication across nodes with primary-secondary replication within a node, where several instances of the database are running on a single machine. This approach deals with the lack of scalability of databases on multi-core systems while minimizing communication costs that ultimately results in an overall improvement of the services. Results show that the proposed solution is able to scale as the number of nodes and clients increases. It also shows that the solution is able to take advantage of multi-core architectures.

Keywords: Distributed Systems, Macro-components, Concurrency, Replication

Resumo

Nos últimos anos, várias abordagens foram propostas para melhorar o desempenho das aplicações em máquinas *multi-core*. No entanto, explorar o potencial dos processadores *multi-core* continua a ser uma tarefa complexa para a maioria dos programadores. Um *Macro-componente* é uma abstração que tenta resolver o problema permitindo explorar o poder das máquinas *multi-core* sem que para isso seja necessário modificar os programas. Um Macro-componente encapsula diversas implementações da mesma especificação. Isto permite tirar partido do melhor desempenho de todas as operações e/ou distribuir a carga pelas réplicas disponíveis, enquanto mantem-se o custo da contenção e sincronização no seus mínimos.

Em aplicações de mundo real, depender de apenas um servidor para fornecer um serviço, leva a uma tolerância a falhas e escalabilidade reduzida. Para abordar este problema, é comum a replicação de serviços em múltiplas máquinas. Este trabalho aborda o problema de suportar tal método de replicação, explorando ao mesmo tempo o poder das máquinas *multi-core*.

Para este fim, propomos a replicação de Macro-components num *cluster* de máquinas. Nesta dissertação apresentamos o desenho de uma solução *middleware* para atingir o objetivo proposto. Utilizando o *middleware* de replicação implementado, conseguimos criar um sistema replicado de Macro-componentes focado em bases de dados em memória que se sabe ter problemas de escalabilidade em sistemas multi-núcleo. A solução proposta combina replicação *multi-master* pelos diferentes nós de uma rede, com replicação primário-secundário num nó, onde várias instâncias de uma bases de dados se encontram a executar numa só máquina. Esta aproximação, aborda o problema da escalabilidade das bases de dados em sistemas multi-núcleo enquanto minimiza o custo de comunicação que por sua vez melhora a escalabilidade geral dos serviços. Resultados mostram que a solução consegue escalar com o aumento de número de nós e clientes enquanto tira partido de arquiteturas *multi-core*.

Palavras-chave: Sistemas Distribuídos, Macro-componentes, Concorrência, Replicação

Contents

List of Figures

List of Tables

Listings

1

Introduction

1.1 Context

With the number of cores per CPU increasing at a very high pace, there is an ever stronger need to use multithreading to fully explore the computational power of current processors. But creating parallel software that truly takes advantage of multi-core architectures is not a trivial task, even for an experienced programmer [Asa+09]. The basic approaches for parallel programming are known to be error prone (e.g., locks), and hard to reason about, due to their low level nature [Har+05]. In the last few years, researchers have been addressing alternative abstractions for concurrent programming, such as *Software Transactional Memory* [Sha+95]. An alternative and complementary approach being developed is the *Macro-component* [Mar10], which consists in several implementations of the same specification encapsulated in a single component. The Macro-component offers an abstraction where the programmer only needs to make a call to the Macro-component interface, and the runtime is responsible for performing the calls to the internal implementations and deal with concurrency issues.

The solutions presented before were designed to run in a single computer. However there is an urge to start creating services with increased availability and dependability. To address these issues, replication and distribution are a common technique. Thus, research has focused on extending solutions for concurrent programming to a distributed environment that includes several machines running in a cluster [Kap+12; Cor+12].

Replication refers to data (or component) redundancy, where two or more copies exist in the system. Redundancy is important in order to provide some degree of availability and fault-tolerance: should a given machine fail, if it is replicated elsewhere, then it is

possible to access the same set of information. Note that this definition says nothing about different replicas cooperating with each other. It only specifies the existence of more than one unit (data or component), with the same content in the system.

Distribution on the other hand, refers to individual machines, potentially not in the same physical space that contribute to the system's functionality. For instance, distribution may help balancing the workload across replicas, or even use each resource in parallel in order to increase the system's performance. Distribution may help keeping failures independent and allow for increased availability, provided that non-faulty replicas are able to keep the service correctly running, which would not be possible in a non-distributed environment.

Together, distribution and replication provide the basis for fault-tolerance and availability. With distribution, different replicas cooperate to provide better service, and with replication they can further cooperate, for instance, tolerating a failure of a node transparently to the client by using a different replica (that is able to process the request) or even provide load balancing features, by distributing the requests among the replicas.

1.2 Problem

Most of the current systems or services are supported by distribution and replication in order to achieve fault-tolerance, availability or even improve performance. Though, nowadays, most of the machines that run these system are multi-core machines. Therefore it is important for these services/systems to take advantage of the multi-core architectures provided by the machines to fully explore the machine's computational power.

To this end, one possible solution is to use Macro-components, that is a centralized solution that explores the power of multi-core architectures. A Macro-component is a software component designed to provide improved performance over standard components, by taking advantage of diverse component replication techniques. To achieve this, a Macro-component typically encapsulates several diverse implementations of the same specification.

To combine Macro-components with distribution we need to replicate the Macro-components across a set of nodes. As the Macro-component has several internal replicas, the challenge is knowing how to deal with replication across nodes with the local replication imposed by the Macro-component. The problem here is how we should present the replicas encapsulated by the Macro-component to the entire distributed system as it may be expensive to present each one of the replicas as a different entity to the system.

Furthermore, how to accomplish the replication of Macro-components in a relevant context such as databases, since they are used in many different contexts and are known to have scalability issues on multi-core architectures.

1.3 Proposed Solution

In this work, we propose *Distributed Replicated Macro-components*, that is a solution for systems that need replication for load distribution and/or fault-tolerance while at the same time exploring the computational power provided by multi-core architectures.

To reach our goal, we have extended the current Macro-component solution with a *middleware* (the replication component) that deals with distribution and replication protocols. The *middleware* itself provides a group communication system that allows the implementation of several different replication protocols for the variety of contexts that it may be used.

In order to keep replication as less expensive as possible, we took advantage of the Macro-component structure, i.e, having many internal replicas and reorganized them. Our approach was to use a primary replica in a Macro-component that is responsible for updates and their propagation to other replicas in the Macro-component. Replication across multiple Macro-components is implemented using a multi-master approach, where updates can be executed in any replica - in this case, for each Macro-component, only its primary. This way only the primary replica will participate in the replication protocol, while the secondaries are "hidden" in a multi-level approach. With this we combine a multi-master (replication across nodes) approach with a primary-secondary approach (local replication) in each local node.

1.4 Contributions

This dissertation presents the following contributions:

- the definition of a generic *middleware* that is able to distribute and replicate any Macro-component;
- a novel approach combining multi-master at network level with primary-secondary locally in order to reduce communication costs in replicated databases;
- evaluation of the proposed solution.
- presentation of an article related to this thesis as a communication in the Simpósio de Informática (INForum) 2013.

1.5 Outline

In this chapter we gave a brief introduction of our work and showed some of the problems that we are trying to solve and the motivation behind it. Afterwards we presented our proposal followed by the contributions. The remainder of this document is organized as follows.

In Chapter 2 we present the related work that is relevant to this thesis. We start by introducing distributed software transactional memory that presents itself as an alternative to our work. We proceed by exploring replication strategies that are commonly applied in database systems, showing their advantages and disadvantages and some optimizations over these protocols. We then present how these can be applied, by presenting systems that implement those strategies. We proceed by presenting state machine replication technique, which is a more generic form of replication, that helped further understand replication and its implications. Lastly we present some examples that show how to apply this method of replication.

In Chapter 3 we present an introductory chapter, that details a bit more on the Macro-component concept and some of the possibilities of such component. We proceed by presenting a concrete example that is focused on in-memory databases.

Continuing to Chapter 4, we start by explaining in a generic way how our solution was built and all of its components. Afterwards, inspired by the example system presented in Chapter 3, we extended that solution and made it work in the distributed setting, i.e, a distributed system of in-memory databases.

We proceed to Chapter 5 where it is presented the evaluation of our solution. Finally in chapter 6 concludes this dissertation with an overview of its main points and some of the future work directions.

2

Related Work

In this chapter, we will describe previous work, and some aspects that should be introduced in order to help the reader contextualize and better understand the concepts as well as a hint for the direction which we will follow.

In Section 2.1 we introduce *distributed software transactional memory*, that presents itself as an alternative approach to our work, and exhibits some similarities such as memory consistency and it may be source of inspiration to our work. We focus on the certification protocols created in this context.

In Section 2.2, we introduce *database replication* and some questions that arise in order to implement reliable replication protocols and strategies. We proceed by presenting an abstract protocol, followed by the strategies that a protocol may implement. We also show how *atomic broadcast* can be used, instead of *distributed locking* to implement concurrency control, and its main advantages.

Then in Section 2.3, we introduce the concept of *state machine replication*, and some basic assumptions and protocol implementation to give the reader a basic introduction. We analyze previous systems that contributed with new protocols in this context, and present some advantages and disadvantages of these contributions.

2.1 Distributed Software Transactional Memory

Developing in a parallel environment is increasingly complex, as the programmer has to reason about multiple threads at the same time. This makes developing parallel software more prone to errors. In addition to this, the traditional methods for concurrency control are known to limit parallelism (critical sections) and therefore hurting performance.

Database systems already tackles the problem of parallelism for years, by hiding the parallelism control and complexity inside transactions. Inspired by this model *software transactional memory* (STM) [Sha+95] was introduced to the programmers as an abstraction that simplifies the creation of parallel programs. This abstraction is based on the concept of memory transactions that are similar to database transactions. When using software transactions, programmers must encapsulate the code that accesses shared data in a transaction.

STM systems usually provide atomicity and serializability properties [Sha+95]. The STM runtime guarantees that the transaction properties are observed by concurrent executions of transactions running in the same system.

Similarly to STM, *distributed software transactional memory* (DSTM) has been proposed as an alternative for distributed lock based concurrency control. Only recently DSTM started to have attention in the community, in order to enhance performance and dependability of applications running in clusters. Similarly to databases, DSTM presents some of the challenges that arise when going to a distributed setting: where will data be located? how will data be distributed? how will protocols keep data consistent?

We will now address the consistency aspect, as it has been the area where novel and improved approaches to provide consistency have been surfacing. In particular certification based protocols have been target of active research, because certification protocols are considered as a suitable alternative to traditional concurrency control mechanisms.

2.1.1 Consistency

Most of the overhead when executing transactions is on the consistency algorithms. This characteristic is also borrowed from the database world, and naturally the literature on database inspired the creation of consistency algorithms for DSTM. When using certification based protocols, a transaction runs in the local replica and the execution is certified at commit time for correctness. The advantage of certification protocols is that synchronization is only necessary at commit time. For read-only transactions remote synchronization is not needed, due to the fact that it is possible to localize transaction execution [Agr+97; Kem+98].

These protocols however, heavily rely on *group communication systems* (GCS) that usually provide a *Total Order Broadcast*[1] (TOB) primitive [Déf+04], which is used to disseminate transactions at commit time. GCS provide two primitives: *a*) TO-broadcast(m), which broadcasts message m; and *b*) TO-deliver(m), which delivers message m.

TOB is attractive for certification protocols, because it has the following properties:

- **Validity.** If a correct replica TO-broadcasts a message m, then it eventually TO-delivers m.

[1] Also known as Atomic Broadcast

- **Uniform Agreement.** If a replica TO-delivers a message m, then all correct replicas eventually TO-deliver m.
- **Uniform Integrity.** For any message m, every replica TO-delivers m at most once, and only if m was previously TO-broadcast by $sender(m)$.
- **Uniform Total Order.** If replicas p and q both TO-deliver messages m and m', then p TO-delivers m before m', if and only if q TO-delivers m before m'.

Many protocols take advantage of these properties to guarantee that all replicas receive the read or write set and that the read or write set is received in the same order. There are two main types of certification protocols that appeared in the context of databases: the *non-voting certification* [Agr+97; Kem+98] and the *voting certification* [Kem+98] strategy. We present both types of certification below, as well as some extensions that appeared in the context of DSTM.

Non-voting certification The non-voting certification approach requires that the replica that started the transaction TO-broadcasts the read and write set using a TOB primitive. This means that all replicas are able to validate and abort or commit the transaction without further communication (replicas have the read and write set). The decision will be the same at all replicas.

Voting certification The voting certification approach relaxes the TOB primitive requirements and only the write set is TO-broadcasted to the remaining replicas. The write set is usually smaller, thus this approach explores the trade-off of exchanging smaller messages at the expense of an extra round of communication. Since only the replica that started the transaction has both the read and write sets, it is the only one capable of determining the fate of the transaction. For this reason this replica needs to additionally broadcast the result message reliably, which will be an abort if it detects any conflict or commit otherwise.

Bloom Filter Certification As demonstrated in [Cou+09], the size of the exchange messages greatly affects the performance of the TOB primitive. The *bloom filter certification* (BFC) [Cou+09] is an extension to the non-voting scheme, that reduces the size of data exchanged. A bloom filter is a data structure that is used to check data items for inclusion. This data structures may yield false positives but never false negatives (if it says no, then that item does not exist). The bloom filter is used in transactions broadcast, by encoding the read set in a bloom filter, in which the size of the read set is computed to ensure that the probability of false positives is below a user defined threshold. The greater the size of the resulting encoded read set, the lower the probability of false positives. At the remote site, inclusion queries are executed on the bloom filter to check if any read occurred at the same time of a concurrent write and aborts if so.

Polymorphic Self-Optimizing Certification Although bloom filter certification is a good alternative to the traditional non-voting approach, it is not optimal for all types of workloads. For instance write workloads or workloads where large read sets are generated are not so adequate for BFC. Each type of workload tend to fit best on different approaches of certification protocols. What is critical is to decide how to deal with the trade off between the size of message exchange (non-voting) and the number of steps needed to execute at commit time (voting). To alleviate the developers from executing time consuming profiling tasks, Couceiro *et al.* [Cou+11] proposed a certification technique that implements some of the certification protocols previously mentioned, and at each transaction, using machine learning algorithms, decides which approach suits best for that transaction. These algorithms have in consideration the different parameters that affect the performance of each approach[2]. Machine learning algorithms are used, in order to accurately predict each parameter value, and use the approach for which the sum of the costs is minimal.

We now present a system that leverages replication to also improve dependability. The chosen system uses the bloom-filter certification strategy.

2.1.2 D^{2}STM

Not many STM solutions address the problem of replication in order to increase dependability, being more focused in performance. Couceiro *et al.* [Cou+09], proposed D^{2}STM, that leverages replication not only to improve performance but also dependability. Consequently data is replicated across all replicas.

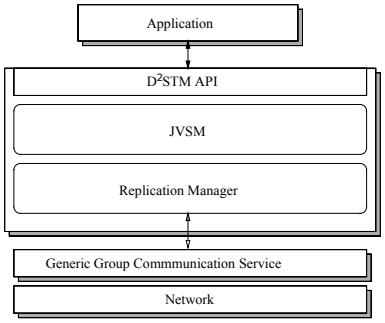


Figure 2.1: D^{2}STM Components (taken from [Cou+09])

As Figure 2.1 shows, D^{2}STM is based on JVSTM [Cac+06]. D^{2}STM also relies on an atomic broadcast primitive, provided by the group communication system.

For coordination, D^{2}STM uses a non-voting certification scheme. Efficiency of these schemes however, are known to be greatly affected by the size of the data transmitted. D^{2}STM overcomes this problem by encoding the read set, on a *bloom filter*, that enables

[2]Message size, transmission time, etc.

high compression rates on the messages exchanged between replicas, at the cost of a tunable increase in the probability of transaction abort due to false positives. This way, D^2STM is able to keep consistency across all replicas with low overhead, and have a relative low probability of aborting due to false positive.

The replication manager is the component responsible for performing the encoding and for implementing the distributed coordination protocol to ensure replica consistency. This component also interfaces with the STM layer, in order to trigger transaction validation and apply the writes sets of remotely executed transactions.

2.2 Database Replication

It is not recent the need for database applications to scale as the number of users increase. Even in the early years this problem started to have some attention in the research community. Along with the performance problem comes the reliability issue. Systems should be able to tolerate some faults and continue to function. Database replication started to emerge in order to deal with both problems. Simple protocols were proposed, but these did not scale in heavy workloads. For these reasons database replication became an important topic, and it is still currently an active area of research.

There are some aspects about replication that should be taken into account. For instance, how should data be distributed? What is the level of consistency required by the system?

In what concerns data distribution, the options are: *full replication* and *partial replication*. Full replication means that all database items are replicated in all replicas, *i.e*, all replicas have a copy of the entire database. This is a straightforward way of replicating data, but it may be unfeasible if the database is very large. Partial replication means that database items are only replicated in a subset of replicas, *i.e*, no replica contains all database items. Partial replication solves the problem of large databases not fitting in a given replica, but decreases availability.

In terms of consistency, a system may employ *strong* consistency levels, where inconsistencies are not allowed to happen and therefore limiting concurrency, or weaker consistency levels, where some inconsistencies are allowed in order to gain performance.

2.2.1 Replication Protocols

In replicated databases, replicas need to know how to interact with each other in order to provide a set of guarantees. This is where replication protocols take over and have the responsibility of managing replicas to keep those guarantees.

Users interact with databases by invoking *transactions* which is the work unit for databases. A transaction is a sequence of read and write operations that are executed atomically, i.e., a transaction either *commits* or *aborts* all of its operations [Agr+97].

To invoke transactions, clients need to connect with replicas. Usually in databases,

a client connects to only one replica, and that replica is responsible for broadcasting the request to the remaining replicas. The reason for this is simple: replication should be transparent to the client.

The way a given replica propagates the requests to the remaining replicas depends on the used strategy. To help describe the different strategies, replication protocols can be composed by the following five generic phases [Wie+00a].

1. **Request:** The system's client submit operation requests to a replica.

2. **Server Coordination:** The replica servers coordinate in order to synchronize the execution of the operations. The main goal of this phase is to obtain an ordering on the requests.

3. **Execution:** The operation is executed on the replicas (master or backups depending on the strategy).

4. **Agreement Coordination:** Replicas coordinate with each other in order to determine if all replicas agree on the result.

5. **Response:** Finally the results are sent back to the client.

An abstract protocol can be built using the five phases explained above, but what usually happens is that a given protocol may use a different approach in a phase that potentially eliminates the need for some other phase.

2.2.2 Replication Protocol Strategies

Database replication protocols can be categorized using two parameters: when the update propagation takes place (*eager vs. lazy*), and where can the updates be performed (*pimary vs. update everywhere*) [Gra+96]. Figure 2.2 shows the identified replication strategies.

In eager replication schemes, the user receives the commit notification only after sufficient replicas in the system have been updated, whereas in lazy replication schemes the operation updates the local replica object returning the result of the commit to the client. The update propagation only takes place some time after the commit.

The eager approach provides consistency in a straightforward way, but the message cost is high as well as it increases the response time by delaying the reply to the client. On the other hand, lazy replication allows for many optimizations, but replica divergence is allowed and so inconsistencies might occur.

In terms of where can updates be performed, in the primary copy approach, one replica is responsible for initially executing the updates. Afterwards the operations will be applied in the remaining replicas. In the update everywhere approach any replica is allowed to receive the updates, which will be later propagated to the other replicas.

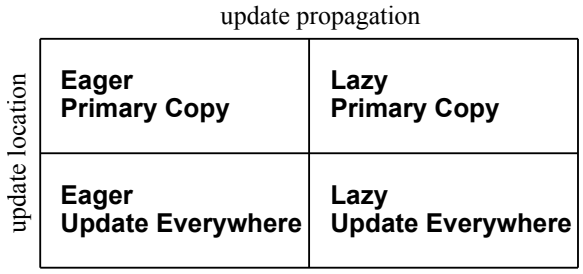


Figure 2.2: Replication strategies in databases (taken from [Wie+00a])

The primary copy approach presents a straightforward way of replica control, but introduces a potential bottleneck and single point of failure. The update everywhere approach allows any replica to be updated thus speeding up access time, but potentially, coordination becomes more complicated.

2.2.2.1 Eager Primary Copy

The *eager primary copy* approach requires that update operations first run on the primary replica, and are later propagated to the secondary replicas. The primary will commit, only after it has confirmation from sufficient secondary replicas that the update was performed. One important aspect of transaction replication is the order given to conflicting operations. This ordering is determined by the primary and slaves must obey the assigned order. Figure 2.3 presents schematically the eager primary copy strategy, using the five phases of Section 2.2.1.

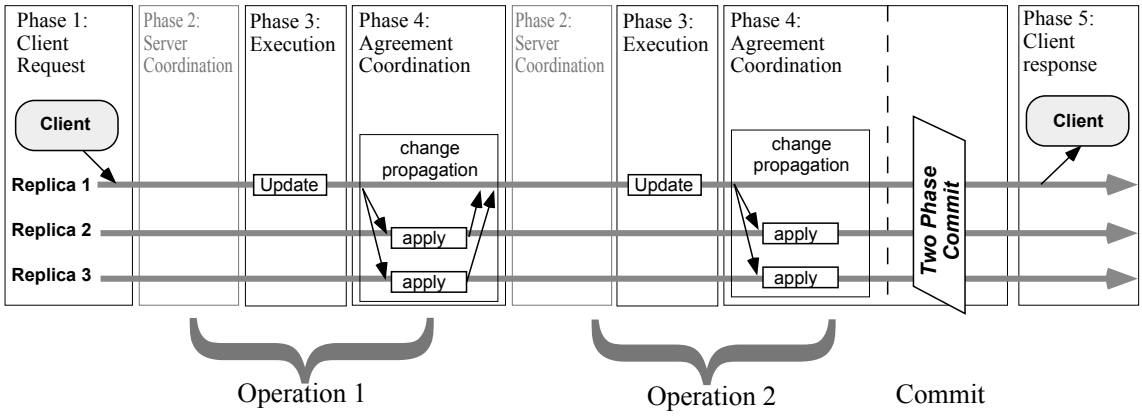


Figure 2.3: Eager primary copy (taken from [Wie+00a])

It should be noted that the server coordination phase is omitted, since execution only takes place at the primary replica.

Execution involves performing the transactions as well as generating the corresponding logs. The logs generated by the primary are then applied by a transaction at the secondary replicas. For this reason a two-phase commit protocol (*2PC*) is needed to guarantee that all replicas agree on the result of the transaction [Wie+00b].

2.2.2.2 Eager Update Everywhere

The approach of Section 2.2.2.1, has the obvious drawbacks of single point of failure and potential bottleneck. The *update everywhere* approach is introduced in order to solve this problem, but at the same time introduces complexity on replica coordination.

Usually two types of protocols are considered for this approach, whether they use distributed locking or atomic broadcast to order conflicting operations.

Distributed Locking In general, transactions contain several operations. In this approach each operation must first acquire a lock in every replica. The locks work like a server coordination phase between replicas. For this reason it is necessary to repeat server coordination and execution phase for every operation in the transaction as Figure 2.4 shows. In the end a *2PC* protocol is used, to make sure that the transaction commits or aborts at all sites.

Although locking provides consistency in a straightforward way, it suffers from significant overhead in replica coordination (acquiring locks at all replicas).

Another issue demonstrated in [Gra+96] is the fact that the deadlock rate is of order $O(n^3)$ in the number of replicas in the system, and therefore system's scalability is limited.

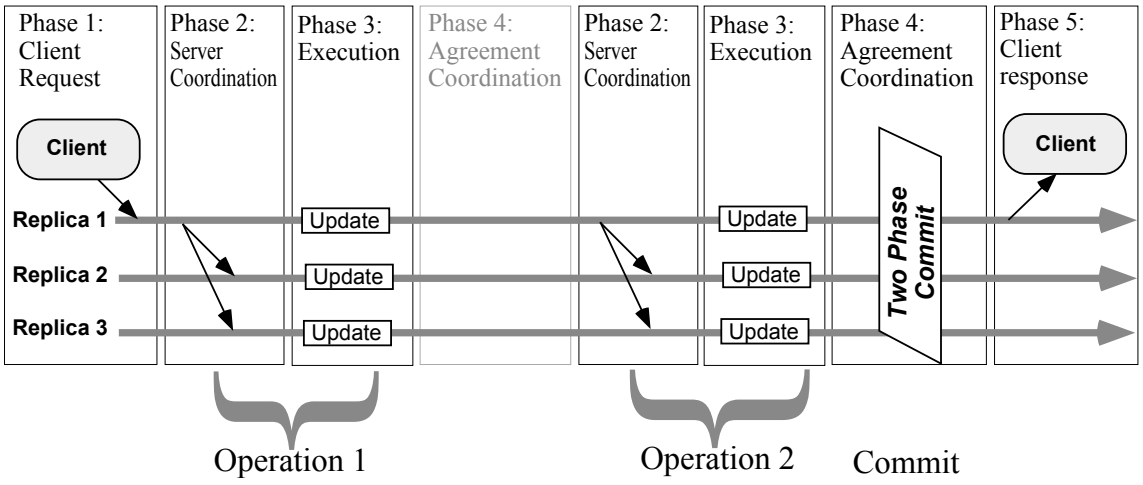


Figure 2.4: Eager update-everywhere (taken from [Wie+00a])

Atomic Broadcast The idea behind using group communication primitives is to use the order guaranteed by the *atomic broadcast* to provide a hint to the transaction manager. What this means is that the *2PC* protocol that is usually employed in the agreement coordination phase is replaced by the total order guaranteed by the atomic broadcast. However, this is only true if a transaction is formed by only one operation. When several operations exist in a transaction, the order provided by the communication primitive has no bearing on the serialisation order that needs to be produced.

Solutions that deal with this problem are known as *Certificate Based Database Replication*. The main goal of certification based protocols is to check if replicas can execute operations in the order given by the atomic broadcast. Figure 2.5 shows a scheme of certification based protocols.

Some examples of certification based protocols are presented in Section 2.1.1.

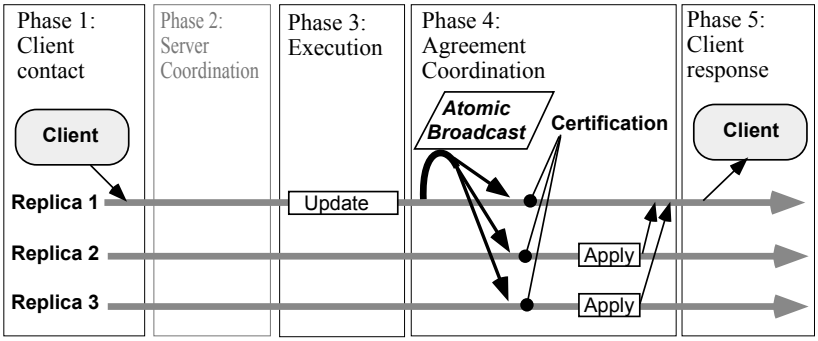


Figure 2.5: Certificate Based Database Replication (taken from [Wie+00a])

Partial vs. Full Replication in Eager Schemes As stated in Section 2.2, full replication presents some advantages, but also presents an important drawback: systems that employ full replication in eager approaches have their scalability limited, because every update operation needs to be applied at every replica. At some point adding replicas will only decrease system's performance.

In what concerns eager approaches, there are some proposals that address the scalability problem of full replication, by employing a partial replication scheme [Ser+07; AIn+08].

2.2.2.3 Lazy Replication

As expected, the eager replication approach has a significant overhead because of the need to commit at the remaining replicas before returning the results to the client.

Lazy approaches try to solve the problem by providing a response to the clients before any coordination takes place.

Lazy Primary Copy A significant difference between the lazy primary copy approach and the eager primary copy is that in the agreement coordination phase, there is no need to run the *2PC* protocol. Since the primary has already provided a response and the update propagation is only performed afterwards, secondary replicas can abort and restart update operations propagated by the primary, without affecting the client. Figure 2.6 shows the scheme of this strategy.

The lazy primary strategy just presented is used for instance, in the *Multimed* system [Sal+11].

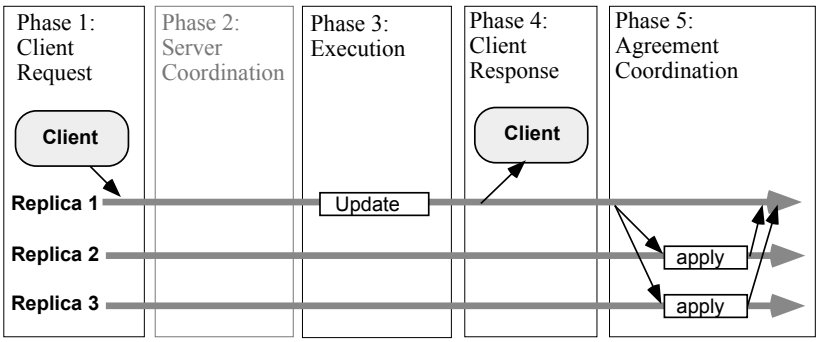


Figure 2.6: Lazy Primary Copy (taken from [Wie+00a])

Lazy Update Everywhere The lazy update everywhere approach is similar to lazy primary copy approach. However the coordination phase becomes much more complex.

It is possible for two nodes to update the same object and race each other to install their updates at other replicas. For this reason it is possible for copies in different replicas, to be stale or inconsistent. In order to address this issue, reconciliation techniques must be used. Atomic broadcast can be used to determine the after-commit-order of conflicting transactions.

As Figure 2.7 shows, the only difference compared to primary copy is the need for reconciliation.

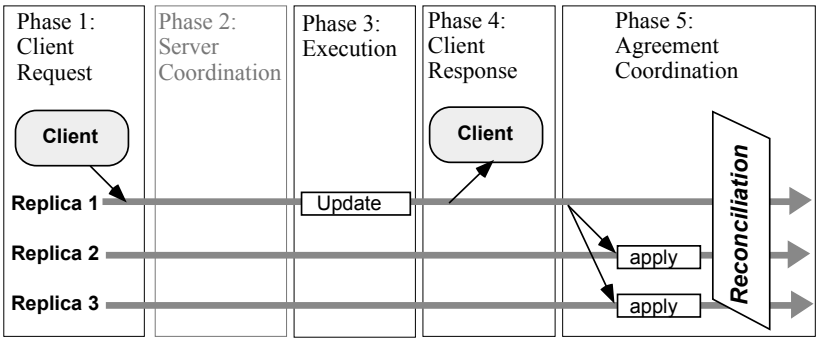


Figure 2.7: Lazy Update Everywhere (taken from [Wie+00a])

These techniques linger in the database replication field for several years. But in practice, they can be applied in many different ways and some approaches do improve some of the drawbacks presented in this chapter. A more comprehensive study in terms of performance focusing on atomic primitives but also covering the lazy strategy is found in [Wie+05].

2.2.3 Systems

We now present some examples of systems that rely on database replication. The selected systems are specially relevant for the work that will be performed: the first work presents a database replication system which strives for load distribution similar to the

concept of the Macro-component; the second work addresses a similar problem but following a different path, namely the use of in-memory databases that due to the fact that they do not incur in disk I/O overhead. And as such the work focuses on how to maximize space usage and thus not being limited by the memory of a single computer. As a side effect work distribution is achieved; The third work presents a database replication system that works in a single multi-core system in an approach that has similarities with Macro-components using typical databases; Finally the last work explores partial replication which may be of interest for this work, if space becomes a problem thus becoming necessary to partition data.

2.2.3.1 Ganymed

Data grids, large scale web applications and database service providing pose significant challenges to database engines. Replication is a common solution, but it has trade-offs. In particular, choosing between scalability and consistency. If strong consistency is chosen, scalability is lost, on the other hand if scalability is prefered than it is the consistency semantics that must be relaxed. Ganymed [Pla+04] presents itself as a middleware that provides scalability without sacrificing consistency and avoiding the limitations of existing approaches.

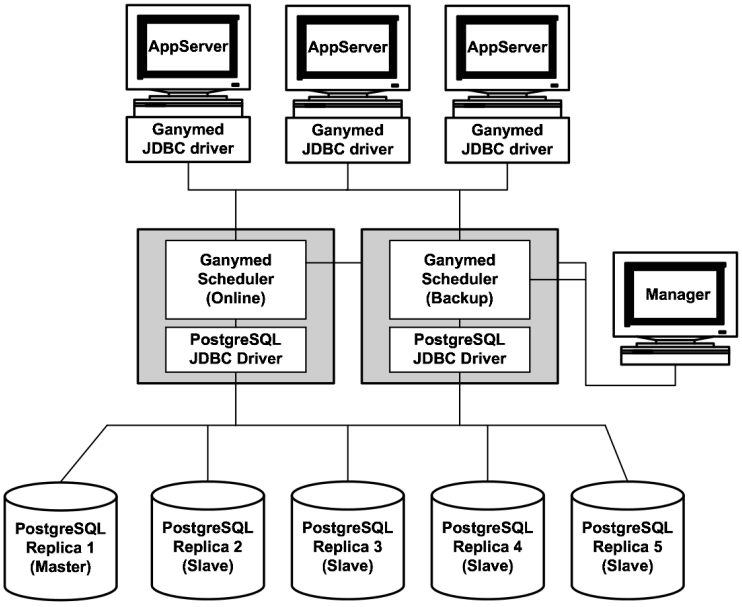


Figure 2.8: Ganymed architecture (Taken from [Pla+04])

Ganymed is built on several components illustrated in Figure 2.8. The main component is the scheduler that routes transactions to a set of snapshot isolation based replicas, using the RSI-PC (*Replicated Snapshot Isolation with Primary Copy*) scheduling algorithm. The clients access the system using a modified JDBC driver. A manager exists monitoring the system for failures, and allows for administrators to change the system's configuration on runtime.

The idea behind RSI-PC is to distinguish between update and read-only transactions, redirecting each type of transaction to a specific replica. In particular, updates are scheduled to a master replica (the primary copy), while reads are sent to any of the other replicas. From the point of view of the client, it seems like it is acessing a single snapshot isolation database. Updates at the master are later propagated to the remaining ones in a lazy manner, but to the client it offers a eager service, thus masking temporary inconsistencies that may occur internally.

To execute transactions the used replicas must offer the transaction isolation levels *serializable* and *read committed*. For update transactions RSI-PC is able to support serializable and read committed, on the other hand read transactions can only be executed in serializable mode. In more detail, transaction execution proceeds as follows:

Update transactions Update transactions are directly forwarded to the master replica. The scheduler must know the order of commits of these transactions. After each successful commit and in order, the scheduler propagates the corresponding write set to the remaining replicas (slaves) and makes sure that the slave replicas apply the write set in the same order as the corresponding commit occurred on the master replica. The scheduler also keeps a global version number, which represents the number of commits done so far, and is used to determine when a slave replica is up to date.

JDBC driver does not have support for write set extraction, which is needed by the master replica. To deal with that Ganymed uses triggers. To avoid adding an extra method only for write set extraction, the write set is inserted into a predefined table. This way standard JDBC invocations can be use to retrieve the write set, for instance by using a standard SELECT query.

Read transactions When a read only transaction is received, the scheduler is able to redirect to any of the slave replicas. If the chosen replica is out of date, the scheduler must delay execution until all changes are applied. For clients not willing to wait, they can either redirect request to the master, or set a acceptable staleness threshold.

Using TPC-W to evaluate the solution, Ganymed achieved almost an linear improvement over the standard component (PostgreSQL), in both throughput and response time to the client.

2.2.3.2 Sprint

High performance and high availability data management systems have traditionally relied on specialized hardware, proprietary software, or both. Hardware has become affordable, but software on the other hand, still is an obstacle to overcome. Recently in-memory databases (IMDB) started to emerge as an alternative to traditional databases and with higher performance. Applications accessing an IMDB are usually limited by the memory of the host machine.

Sprint [Cam+07] presents itself as a data management middleware that replicates an IMDB across a set of share-nothing data-servers, by partitioning the data into segments. This way, applications are now limited by the aggregated memory of the machines in the cluster.

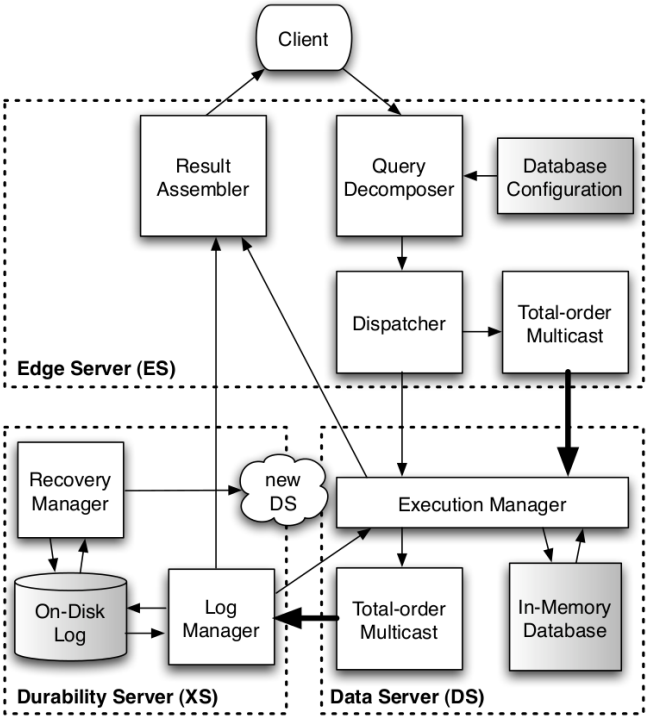


Figure 2.9: Sprint architecture (taken from [Cam+07])

To build such middleware, it was necessary to divide it in several sub-components each with different responsibilities as shown in Figure 2.9. Sprint distinguishes between physical servers and logical servers, which is the software component of the system. There are three types of logical servers: edge servers (ES), data servers (DS), and durability servers (XS). DS's run an IMDB and execute transactions without disk access. It is the durability server's responsibility to make changes durable. The edge servers are the frontend for the clients. Durability servers are replicated to provide high availability whereas data servers are replicated for both availability and performance.

In Sprint there are two types of transactions: local transactions and global transactions. Local transactions access only to data on the same sever. Global transactions on the other hand access data in multiple servers. This distinction exists, because data is sharded across data servers, and one query may access several data servers.

The execution of local transactions is straightforward: every SQL statement received from the client is forwarded to the corresponding DS for processing and results returned to the client. When a transaction is or becomes global, the ID of the transaction is multicasted to all DS's. The multicast primitive induces a total-order on global transactions, used to synchronize their execution and avoid deadlocks. When a conflict is detected the

assigned ID is used to determine which transaction is aborted, notifying the associated edge server. The conflict definition in this system is relaxed. It is only required that two transactions access the data on a data server, and one of them updates the data. This may lead to false conflict cases, but maintains the serializability consistency level.

To terminate a read transactions terminate when the edge server receives an acknowledgment from every involved data server. If it suspects some failed, it aborts the transaction.

Terminating write transactions is more complex, as it is necessary to guarantee that the committed state will survive server failures, and thus XS servers also participate. In this case an optimized version of PAXOS [Lam98] is used as means to implement a total order multicast. If the vote is to commit, than all servers will come to the same decision and the DS will commit against their local IMDB. Consequently the durability servers also receive these votes and make changes durable in disk.

The system was evaluated using MySQL in in-memory mode and TPC-C. Results show that when there is enough data to fill the memory of several data servers, Sprint manages to outperform the standalone server in throughput and response time by a factor of up to 6 times. This may be due the fact that transactions may be executed in parallel in different places. This situation is maximized when the data is partitioned in a way that maximizes the possibility of parallel execution, i.e, maximum distribution of transaction load.

2.2.3.3 Multimed

Most commercial database systems are designed to run on a single CPU machine, and are optimized for I/O bottlenecks. For this reason, contention is high when the number of threads increases and consequently these systems fail to scale properly on multi-core machines.

Salomie *et al.* [Sal+11], proposed a system (*Multimed*), that address these issues by relying on database replication in a single multi-core machine.

Multimed uses each core in a multi-core machine, as a pool of distributed resources. On each core, a database engine is running and communicates with other engines, using distributed systems strategies in order to replicate data.

Multimed has a core (or replica) that acts as the master. This replica is responsible for receiving the updates and execute them and later propagating the changes to the slave replicas (or *satellites* in Multimed's terminology). A Read-only transaction can execute directly in a slave replica, thus distributing the load among replicas. The presented scheme is very similar to the lazy primary strategy (Section 2.2.2.3).

Besides the database replicas, Multimed has an additional key component: the dispatcher. The dispatcher has the responsibility of redirecting the requests to the correct

replica, because the system may employ a partial replication scheme. In this case, the dispatcher has the needed information statically, so it redirects requests to the "right" computational nodes (satellites or master). To select the replica to execute the transaction, the dispatcher takes into consideration the type of transactions and for read-only transactions it selects the slave that is up-to-date with lower load.

With this implementation strategy, Multimed is capable of reducing contention costs, and correctly scale in multi-core machines. Tests showed that Multimed scales linearly in read intensive tasks as the number of cores increase, contrary to MySQL and PostgreSQL. On write intensive tasks Multimed hit the limit much faster, but still kept stable performance, whereas, on the standalone versions, performance decreased rapidly.

2.2.3.4 Partial Replication with 1CSI

As previously mentioned, when full replication is used, all replicas need to perform the update in order to keep all replicas consistent. This alone is enough to put a ceiling on scalability. Along with the full replication problem is the fact that most databases that employ full replication, look for serializability, and consequently, all combinations of write/write or read/write conflicts may be ruled out.

Serrano *et al.* [Ser+07] proposed a solution, with partial and *1-copy-snapshot-isolation* (1CSI), to address the scalability issue.

The protocol proposed employs a eager update everywhere strategy (Section 2.2.2.2), and it is designed to work with databases that employ snapshot isolation as their concurrency control mechanism. Snapshot isolation has the advantage that read transactions never abort, so only write/write transactions are checked for conflicts.

Clients start by issuing requests to a replica that has at least the object of the first operation. If that replica has all the objects that the transaction accesses, then the transaction is executed in that replica, otherwise, the transaction is forwarded to another replica with the required data. When the coordinator forwards the transaction, it also sends the timestamp of the beginning of the transaction in order for the transaction to read from the same snapshot version in all replicas. It may be the case that before the transaction is forwarded, some objects in the current replica were already modified. The transaction should be able to see its own writes, which won't be available in the forwarded replica. For this reason, the changes produced by the transaction on the coordinator are applied at the forwarded site before continuing execution.

Validation is the next step of this protocol. The transaction is multicasted to all replicas (in total order) and validated in all replicas. This way, replicas are able to validate transactions with the necessary information to detect conflicts.

2.3 State Machine Replication

Another technique that has been around for years is the *state machine approach*. The state machine approach is a general method for implementing fault-tolerant service by replicating servers and coordinating client interactions with server replicas [Sch93].

The key idea of state machine replication is for replicas to deterministically process the same sequence of requests so that replicas evolve through the same sequence of internal states and produce the same sequence of outputs. As consequence state machine replicas must implement the same specification and execute deterministic operations. Otherwise replicas can diverge and the system becomes inconsistent.

An advantage of state machine replication is that it transparently masks failures to the clients because, all replicas provide a response and by voting the client can determine the correct reply. But depending on the type of failures supported by the system the condition to accept a reply may be different, which we address this issue below.

2.3.1 Fault Tolerance

A replica is considered *faulty* once its behavior is no longer consistent with its specification. Usually, two representative classes of faulty behavior are considered:

- **Byzantine Failures.** The replica can exhibit arbitrary and malicious behavior, with the possibility of cooperating with another faulty replica.

- **Fail-stop Failures.** In response to a failure, the replica changes to a state that allows for other replicas to detect that a failure has occurred and then stops. A broader case of the fail-stop failures, are the *crash* failures, where the replica simply stops, without any kind of signal or state that allows other replicas to detect that it has crashed.

For most applications, assuming fail-stop failures (or crash) is acceptable, but research work is often focused on byzantine failures, because byzantine failures can be the most disruptive to the system.

A system consisting of a set of distinct replicas is t fault-tolerant if it satisfies its specification, provided that no more than t of those replicas become faulty during some interval of interest [Sch93].

In the presence of byzantine failures, a system that implements a t fault-tolerant state machine, must have at least $2t + 1$ replicas. In this case, the output will be the output produced by the majority of the replicas. This is because with $2t + 1$, the majority of the outputs remains correct even after t failures, assuming all replicas provide a response. If there are no assumptions about replicas responses then $3t+1$ replicas are needed [Bra+85] to provide a t fault-tolerant system.

When the system experiences fail-stop failures, then $t + 1$ replicas suffices, because only correct outputs are generated by fail-stop replicas and thus any of the responses provided can be accepted, i.e., the client can accept the first reply it receives.

The number of replicas that reply to a request is not itself enough to guarantee its correctness. This is because we have to guarantee that each of the replies went through the same state transitions, i.e, they all processed the requests in the same order. The key to achieve this is to have *replica coordination*, which itself can be decomposed into two requirements:

- **Agreement.** Every non-faulty state machine replica receives every request.
- **Order.** Every non-faulty state machine replica processes the requests it receives in the same relative order.

The agreement requirement deals with the interaction between the clients and the state machine replicas, while the order deals with the behavior of the state machine replicas when receiving several requests, possibly from different clients. Systems typically use the following two techniques two guarantee these properties.

Primary Replica A replica is designated as the primary and it has the responsibility of ordering and propagating updates to other replicas. This approach is very simple and it only becomes more complex to deal with a faulty primary. In this case replicas must agree that the primary is faulty and elect a new primary.

Atomic Broadcast The problem of agreement and order has already been discussed in Section 2.2, where atomic broadcast has already been proposed for some protocol strategies. Figure 2.10 shows how atomic broadcast can be used in the context of state machine replication, namely in the interaction between the clients and the state machine replicas. Clients can propagate their requests using atomic broadcast, and the agreement and order requirement will be satisfied.

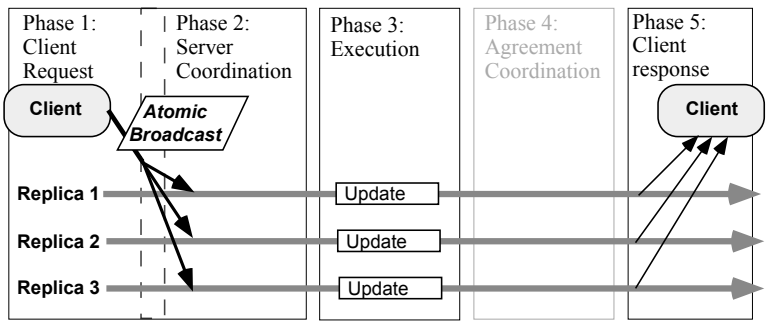


Figure 2.10: State Machine Approach (taken from [Wie+00a])

2.3.2 Systems

We now present some example of systems that rely on state machine replication. These systems are relevant to our work because state machine replication is a technique that favors the current Macro-component execution model. But state machine replication presents some drawbacks, such as the requirement of deterministic execution or the tendency to increase the latency effects on requests. The chosen systems try to address some of the problems. The first one comes more as a historical reference, as several other approaches are based on this one. The second one tries to solve the problem of latency on high latency networks by improving throughput on these conditions. The third shows how to use weak consistency levels on state machines, which can be interesting for Macro-components. The last system presented focus mainly in how to take advantage of multi-core machines, which is the main goal of our work.

2.3.2.1 PBFT

Protocols to reach agreement that tolerate byzantine failures are known as *byzantine agreement protocols*. Castro *et al.* [Cas+99] proposed PBFT, that is a byzantine tolerant protocol for state machines [Sch93]. The problem at the time, was that most earlier work (e.g., [Mal+97; Gar+93]), depended on synchrony, i.e., relied on known bounds on message delays, or were too inefficient to be used in practice. PBFT (Practical Byzantine Fault Tolerance) was the first solution proposed that did not assume synchronous network and that could be used in practice.

The protocol is able to work in an asynchronous distributed system and supports arbitrary behavior from replicas, assuming that a limited number of faults happen.

PBFT protocol also offers protection against third party attacks like spoofing and replays, by using some cryptographic techniques applied to messages. This way the server and client replicas are able to check if a given message/reply is valid.

PBFT defines a primary replica that is responsible for coordinating operation execution. A request is first sent to the primary where afterwards, it starts a three-phase protocol to multicast the request to the remaining replicas (Figure 2.11). The goal of this protocol is for replicas to agree and be able to prove the order of execution of operations. After execution the replies are sent directly to the client, where it checks if enough replies are valid.

The proposed protocol in order to tolerate up to f simultaneous (byzantine) faults, requires $3f + 1$ replicas, which has been proven to be the theoretical minimal number of replicas for tolerating byzantine failures in an asynchronous environment [Bra+85]. In PBFT protocol the client needs to wait for $f + 1$ responses with the same result to accept a reply.

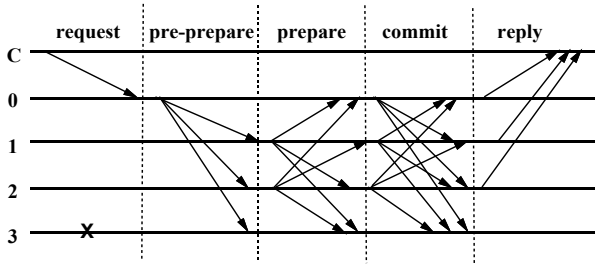


Figure 2.11: Three-phase protocol (taken from [Cas+99])

Results using a replicated NFS service showed that replicated NFS using PBFT protocol, was only 3% slower than the unreplicated version. On the micro-benchmark, the latency of invoking operations (with null requests) is much larger due to the use of public-key cryptography.

PBFT also has the problem of tolerating only f faults in the entire lifetime of the system. To tackle this and the overhead caused by the public-key cryptography, Castro *et al.* [Cas+00], extended their work and proposed a new solution. The extension basically has the goal to act proactively on replicas recovery. The general idea is to "reboot" replicas periodically, thus being able to recover from failures that didn't even had time to manifest their presence and allow the system to react in the usual way. For this reason replicas can discard the use of public-key cryptography and use symmetric keys, because these are also refreshed periodically, and at each reboot. With this approach, the time that the system can handle f failures, is the time between recoveries. After the recovery the system is able to tolerate new f failures.

2.3.2.2 Client Speculation

It is well known that in order to have maximum fault tolerance, replicas should be distributed geographically. On the other hand state machine protocols tend to magnify the latency effect that arises from geographical distribution.

Wester *et al.* [Wes+09] proposed the use of client speculation in order to decrease latency, and contributed with an extension to the PBFT protocol that uses speculation (PBFT-CS).

The general idea of speculative execution is to predict the outcome of a given request based on previous replies. Assuming that errors are rare, and all replicas in the system have the same state, we can say that the first reply received by the client is an excellent predictor of the final outcome. With this assumptions, it's clear that the latency can be reduced, because the client does not need to wait for $f + 1$ equal replies, and is able to continue execution based on the predicted reply.

When a client executes operations speculatively, there is a chance that the first prediction was wrong, and it may be necessary to roll back the changes. To be able to support this correctly, clients must first take some steps when receiving speculative replies. Upon

receiving the first reply, the client should save its current state and then execute further operations based on that reply. This way, in the event that the speculative reply was not an accurate guess of the final outcome, the client is able to roll back on its changes and re-execute the request, now using the correct data. As expected, roll backs must be avoided, as it will hamper system's throughput. For this reason, speculation should be used only, in operations that have predictable results.

Generally speaking, in order for this technique to work efficiently, the closest replica to the client should execute and reply to the client as soon as it receives the request (called the *early reply*). But for agreement protocols like PBFT [Cas+99], it's more elegant if the primary is the one to do this job (Figure 2.12). Consequently the primary should be located near the most active clients in the system to reduce their latency.

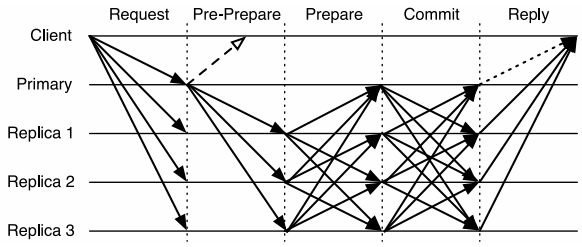


Figure 2.12: PBFT-CS *early reply* (taken from [Wes+09])

This modified version of PBFT differs from the original in several key aspects:

- A client may be sent a speculative response that differs from the final consensus value.
- A client may submit an operation that depends on a failed speculation.
- The primary may execute an operation before it commits.

PBFT-CS was tested against a simple counter benchmark, and compared with the original PBFT protocol and Zyzzyva [Kot+07] which is another agreement protocol that also uses speculation, but on the server side.

In high latency setup, PBFT-CS took 50% less time than the original PBFT and 33% less time than Zyzzyva to complete the benchmark. However client speculation has limited peak throughput under heavy loaded systems, this is due to the lack of resources available and eventual roll backs.

2.3.2.3 Zeno

While most byzantine fault tolerant protocols, aim at providing strong consistency, these lack on high availability properties as the failure of some replicas or network partitions, may cause an entire service to become inoperable.

Many services favor availability over consistency, because they need to provide tight SLA's (*Service-level Agreement*) [DeC+07; Ghe+03]. This makes byzantine fault tolerant protocols undesirable for these systems. Nevertheless, byzantine protocols could be of use even in these trusted environments, as outages have already been experienced by service providers [Ama08], due to internal state corruption, among other examples. These types of failures are well suited for byzantine tolerant protocols.

Singh *et al.* [Sin+09] proposed *Zeno*, a byzantine tolerant state machine protocol based on Zyzzyva [Kot+07], that trades strong consistency for a weaker consistency semantics (*eventual consistency* [Vog09]), in order to provide a highly available state machine.

In order to implement the eventual consistency semantic in state machines, two kinds of quorums were created:

- *Weak quorums*, which consist in the response of $f + 1$ distinct replicas.
- *Strong quorums*, which consist in the response of $2f + 1$ distinct replicas.

Despite the number of replicas that exist in the system, with these two types of quorums, progress can be guaranteed as long as $f + 1$ replicas are available, whereas in traditional (strongly consistent) BFT systems, $2f + 1$ replicas must be available. This is an advantage of Zeno's policy, because even in the presence of network partitions, it is possible for clients to continue issuing requests.

A client starts by issuing a request to all replicas (with the option of using strong quorums). Then the primary assigns a sequence number to the request, and broadcast the request to the remaining replicas. This imposes some overhead on communication because the clients also issues the request to all replicas, but allows backup replicas to verify if the primary did not modified the request. The replicas will execute the request, and if only a weak quorum is required, the replicas immediately reply to the client, otherwise replicas will wait for the request to *commit*, i.e., replicas wait for $2f + 1$ commit replies from other replicas, and only then a reply is sent to the client from the replicas.

The key difference of Zeno's protocol is in the view change mechanism. Since concurrent executions may occur, it is necessary for replicas to be able to merge the divergent replicas. A design choice was to incorporate this change in the view change mechanism. Replicas keep a hash-chain (h_n) of the history of requests, and at each commit or new request (from primary) these histories are compared[3]. These hash-chains are computed as follows: $h_{n+1} = D(h_n, D(REQ_{n+1}))$, where D stands for a cryptographic digest function, and REQ is the request assigned to sequence number $n + 1$ [Sin+09]. It is possible to detect that a concurrent execution has occurred at replicas i and j, if there exists a sequence number n such that $h_n^i \neq h_n^j$.

Whenever a replica has proof that a concurrent execution occurred, it starts a view change. Eventually the new primary will complete the view change, sending the state

[3]histories are sent with these requests from the sending replicas

of all participating replicas in the final message. Replicas will form the new initial state based on the information of this message. First, replicas will start from the highest commit in the message, and then apply all the weak requests where order is determined by a deterministic function that all replicas have. This way, all participating replicas will end up in the same final state, after a view change.

2.3.2.4 Eve

State machine replication requires that replicas execute requests deterministically in the same order. Multi-core servers naturally pose a challenge for this requirement, as requests may be interleaved differently in each replica. This may lead to final divergent states on replicas and the possibility of different results even if no faults occurred.

Eve [Kap+12] tries to integrate concurrent execution in the state-machine replication model, by overcoming the non-determinism problem in multi-core servers by switching from a traditional agree-execute architecture [Cas+99; Dis+11; Wes+09; Cow+06] into a execute-verify architecture. Eve is able to tolerate byzantine faults and it has the possibility of working in a *primary-backup* approach.

Eve creates a *parallel batch* of requests to execute concurrently in each execution replica. Requests inside these batches are said safe to run concurrently, by the *mixer* component, that is the same in all execution replicas. The mixer's job is to parse incoming requests, and determine if there are potential conflicts between the requests, and if so, it puts those requests in different batches.

In order to take advantage of parallel execution, Eve executes first and then verifies, so naturally, the computation is split in two stages: *execution stage* and *verification stage*.

Execution stage In this stage, the requests are sent to the primary, in which it batch requests (no mixer involved in this step) and sends them to the execution replicas, each with its own sequence number. The receiving replicas, implementing the same mixer code, process these batches, and from those batches, extract the parallel batches that can be formed. This parallel batches are then executed in pre-determined order (requests in these batches run concurrently). At the end of each batch, a hash is computed and deterministically inserted into a *Merkle tree* [Cas+03].

Verification stage Since execution stage does not offer guarantees that anomalies did not occur, the verification stage has the job to check if all correct replicas finished in the same state. The root (*token*) of each replica's Merkle tree is sent to the verification replicas, where it is verified if enough tokens match. In the latter case the replicas commit, otherwise they must perform a roll back.

A roll back consists in, undoing all changes to the last committed sequence number, and re-execute the batch sequentially. For this reason the efficiency of the mixer is crucial

to achieve the best performance with this system.

It is easy to verify that this strategy may effectively take advantage of multi-core servers, as long as the mixer, does not put many conflicting operations in the same batch.

One point to refer is that it is not obvious if it is possible to make a replica take the role of both verify and execution replica. If that is not the case then more replicas are required (compared to traditional PBFT) to run the system. This may be a disadvantage compared to other systems.

3

Macro-Component

Our work builds on Macro-components, an abstraction to improve performance and reliability of software components by relying on replication.

Section 3.1 gives a brief description of the Macro-component, and some of the motivations behind the construction of such component.

Next, Section 3.2, details a bit more on how the Macro-component is designed and its functionality. It also covers some topics that were considered while the Macro-component abstraction was being developed and discusses their advantages and disadvantages.

Section 3.3, focus mainly in the overhead that the Macro-component may impose.

Finally Section 3.4 presents MacroDB, that is a concrete implementation of a Macro-component for databases. We start by showing the design of the MacroDB and some of the differences with the generic model. Then we give some detail on how MacroDB works, i.e, how it deals with transactions. And finally some results to show MacroDB's feasibility.

3.1 Overview

Applications are commonly built around a set of software components (e.g, data structures, algorithms, database systems, XML libraries, etc.). These components frequently present a homogeneous interface which hides the underlying implementation (e.g, *HashMap* and *HashTree* both sharing *Map* interface in *Java*). Although all implementations that respect a specification have the same final outcome for each operation, they often differ in terms of performance (time, space, etc.) due to differences in their internals.

A Macro-component is a software component that combines several implementations of a given component specification. These implementations may be different, as long as

they respect the same interface. In the scope of the Macro-component an implementation is called a replica. By encapsulating several replicas, a Macro-component can be used for a number of reasons. The differences in performance of each replica is one of the reasons that lead to the design of the Macro-component. For instance, in a particular application if the workload is well defined it is easy to decide which data structure to use. On the other hand, when the application has a complex flow, it is not trivial to determine what is the dominant workload and which data structure(s) suit best. In this case the Macro-component could be used by invoking all operations in parallel at all replicas that the Macro-component contains. Using this strategy, one can obtain the best performance for all operations, thus improving the overall performance of the application. A scheme of such execution model is depicted in Figure 3.1.

Figure 3.1: Parallel execution of operations in the Macro-component (taken from [Mar10])

Besides improving performance, the Macro-component can be used to improve reliability, by masking software bugs. For instance, to mask a fault in one of the replicas the Macro-component could be composed by three replicas. In this scenario a voting step would be necessary and the majority of equal responses would be treated as correct, while the response of the failed replica would be discarded (which presumably returned an incorrect result).

In this work we will build on a version of the Macro-component that improves application performance.

3.2 Design

To be able to build a Macro-component in a generic way, it was necessary to divide it in smaller components and define the interaction between them. Below we present an abstract scheme of the Macro-component and a brief explanation of each sub-component.

3.2.1 Architecture

Figure 3.2 shows the sub-components that form the Macro-component and their interaction. The usual flow of execution starts by the invocation of an operation in the Macro-component manager. Afterwards, that invocation goes through the scheduler that decides if the operation is sent directly to a particular replica, or to the work queue that delays the execution of the operation to some time later in the future and executes it on a replica or set of replicas.

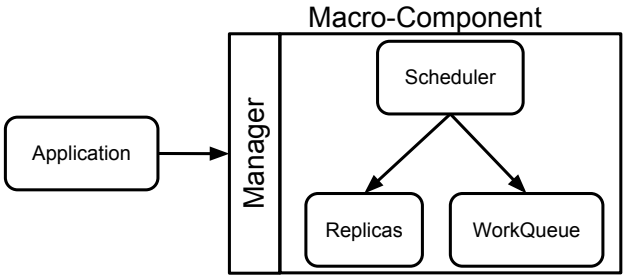


Figure 3.2: Macro-component architecture

Manager The manager is the component responsible for providing the interface of the Macro-component. The interface should be the same as the one provided by the Macro-component's underlying replicas (e.g, *Map* in Java). When an application invokes an operation on the manager, it forwards the operation to the scheduler which is responsible for deciding the operation's next step. The manager is able to submit the operation to the scheduler either synchronously, which will block the application thread until a result is obtained, or asynchronously, which allows the application thread to continue execution.

Scheduler The scheduler is responsible for deciding where the operation goes next. Each operation can be put to execution immediately in a certain replica, or inserted into the work queue where the operation remains until an executor retrieves and executes it.

Replicas Replicas are the implementations of the Macro-component's specification. When an operation is invoked on the Macro-component, the operation ultimately will be executed in one or more replicas, depending on the scheduling scheme. The implementations may differ, for instance to take advantage of the performance differences, or they can be the same and be used for other purposes like avoiding contention in a given replica by distributing the load among all replicas.

Work Queue The work queue is a data structure that stores the jobs that were delayed by the scheduler for later execution. When asynchronous operations exist or when multiple requests arrive, it is likely that the working queue grows to store several jobs awaiting execution. One such case could be when a given replica is propagating updates to the remaining ones. In this case, the update operations remain in the queue until certain executors retrieve the operation, and update the corresponding replica.

One thing to note is that the structure used to store the operations may help in the operations' scheduling. For instance, a FIFO data structure inserts the last request at the end of the data structure, which is convenient for a scheduler that follows the FIFO strategy.

Executors The executors are simple worker threads. These threads execute in a loop, polling the work queue for new jobs. When an operation is inserted into the work queue,

an executor retrieves the operation, processes it, sets the result and signals the scheduler that the operation has completed.

The threads are created when the scheduler is initiated. In our case, the number of threads is fixed with one thread per replica, i.e, each executor is responsible for one replica and all requests in the work queue for that replica. By keeping the number of threads constant, the overhead is minimized compared to a dynamic thread allocation scheme.

3.2.2 Execution Strategies

Having described the components that form the Macro-component, we will now show the studied techniques to leverage these components towards performance, when the Macro-component was developed.

One way to take advantage of the component(s), is by taking advantage of the scheduler that gives the flexibility of applying different operation scheduling strategies. To cover this topic, we must distinguish between two types of operations: read operations and write operations. Write operations are defined as operations that can affect the result of subsequent operations on the replica they are executed on. Read operations, on the other hand, do not affect the results of following operations. Defining the read and write operations in this manner, allows the existence of read operations that changes internal state. For instance, a data structure with promotion changes its internal state to make the lookup for a frequent value more efficient, but it does not affect any subsequent read results.

We are going to focus on read operations, because unlike write operations, they don't need to be executed in all replicas thus enabling the use of some interesting scheduling strategies. Besides the strategy of parallel execution at all replicas previously mentioned, one can choose another path, distributing the requests across the replicas [Pla+04; Eln+06]. We can identify three strategies: read-all, read-one and read-many. Of course for these to work, we assume that the replicas reflect all previous updates at a given moment. We briefly explain these strategies below.

Read-All The read-all strategy is the simplest one of the three. The scheduler just needs to redirect an invocation to all available replicas and wait for the first to return. We can see this strategy as a race between replicas to execute the operation first. Figure 3.1 illustrates this strategy. Using this strategy, the overall performance of the application is improved by ensuring the best performance for all operations. However, this is true only if the system experiences light load and allows for one operation to end before the next invocation. Otherwise, since the replicas must process every operation, it is possible for this strategy to hold back replicas, keeping them busy for too long. This may lead subsequent operations to execute in slower replicas for that type of operation, and ultimately the possibility of performance loss. Figure 3.3 illustrates this situation for a Macro-component with two replicas.

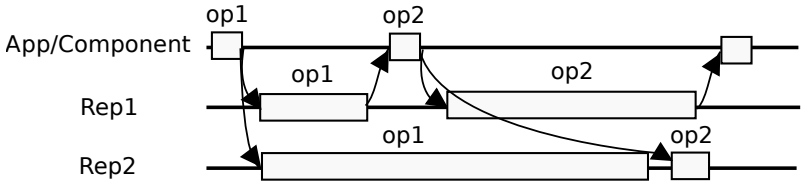


Figure 3.3: Read-All strategy problem (Adapted from [Mar10])

Nonetheless, this strategy presents some advantages. One is its simplicity as it requires no special knowledge about the operations or about the Macro-component replica implementations, allowing this strategy to be applied automatically to any Macro-component.

Another advantage occurs when operations have unpredictable execution times, i.e, there is no best replica for a given operation: in one invocation one replica may be faster than the other, but in another invocation of the same operation a different replica has the best performance. When the situation just described happens and using the read-all strategy, it is possible for the Macro-component to have a better overall performance than either of its replicas.

Read-One In read-one strategy, the Macro-component's scheduler attempts to direct each read operation to a single replica. Optimally, if several replicas exist having different performances, the replica for which the operation is directed should be the one with the best performance for that operation. Figure 3.4 illustrates this strategy. Comparing to Figure 3.3, it seems the "busy" replica problem is solved. Besides that, this strategy allows for replicas to execute different operations in parallel on different replicas.

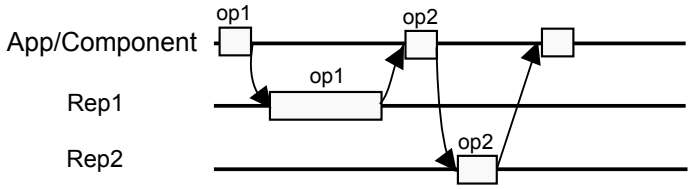


Figure 3.4: Read-One strategy (Adapted from [Mar10])

However, it is still possible for this strategy to suffer from problems similar to the "busy" replica problem explained in the read-all strategy. Assuming a given replica is not able to execute operations in parallel, then every request directed for that replica only starts after the last request has completed. If the redirection policy is to redirect operations to their best replica, or some sort of other static assignment strategy, this may cause operations to wait some time before execution because the replica may be overburdened, thus not fully exploring the parallel possibilities of a multi-core system. One way to deal with this, is to direct some operations to other replicas when the overburden situation is detected. Some care is needed if such system is to be implemented as it may introduce overhead in the runtime.

An important thing to note about this strategy, is that it is possible to have an arbitrary

complex system (static or dynamic) to distribute requests among replicas that may help take advantage of the underlying system, thus improving application performance.

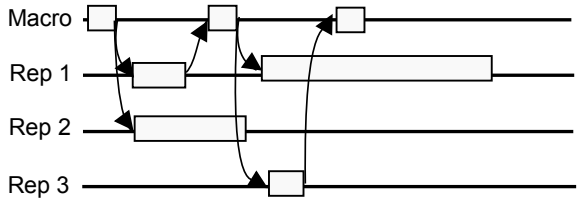


Figure 3.5: Read-Many strategy (Adapted from [Mar10])

Read-Many Read-Many strategy presents itself as a middle ground between Read-All and Read-one strategies and it directs a given operation to a subset of replicas, as shown in Figure 3.5. As usual in hybrid solutions, it takes advantage of the best features of each approaches, but also presents some drawbacks.

One drawback is the need for a larger number of replicas to really distinguish itself from the previous strategies (Read-One and Read-All). If the Macro-component only contains two replicas, a subset can only be formed either by one replica or both replicas. In the first case, execution would be equivalent to a Read-One strategy. In the second case, execution would be equivalent to a Read-All strategy. From this, we can conclude that at least three replicas are required for the Read-Many.

Another drawback is that like Read-All strategy, some operations may lead to waste of processing resources when replicas are competing with each other. But contrary to Read-All strategy, it does not apply to all replicas, only the subset in use, leaving the remaining ones free to execute any other operation in parallel. Furthermore, this strategy may resolve the problem of Read-One strategy: by executing in a subset of replicas, if one replica is overloaded, there is a chance that the remaining ones (where the operation was directed) are less overloaded or free, and process the operation much faster.

3.3 Discussion: Overhead

Although the Macro-component is built to take advantage of multi-core system, the design of such component does not come for free and poses some technical challenges. Being the Macro-component a kind of "proxy" between the applications and the underlying implementations, it will always exist some extra computation time in each individual operation. Next we point some of the sources of overhead in the Macro-component in terms of time and space.

Time overhead The code of the Macro-component itself introduces some overhead in the execution time. Part of this overhead is due to synchronization mechanisms. The need for such mechanism may be for instance, in the case where a given thread needs to

wait for an outdated replica to be brought up to date before any other operation is executed in that replica. Related to that is the housekeeping done by the component to keep information of whether the replicas are up to date or not. Synchronizing updates may also be necessary, for example, to guarantee that the ordering of updates is maintained keeping replicas consistent.

Memory overhead Memory overhead comes from the design of the Macro-component, that typically is composed by more than one replica. Theoretically implementations are usually independent and share no data. Consequently, it is possible for the memory consumption growth of the Macro-component, to be linear with the number of replicas. In practice, as many software components act as containers, data objects stored in the components may be shared among the multiple replica, thus minimizing the space overhead (as discussed later in practical examples).

This does not present a serious problem, as long as the system is able to keep all data in main memory. On the other hand if the used memory by the Macro-component is larger than the system's memory, it will be forced to swap from memory to disk and vice-versa, thus increasing the time overhead already introduced by the Macro-component.

One way to solve the memory overhead problem, is to dynamically create new replicas. In other words, the Macro-component could start with only one replica, and create new replicas as necessary (e.g, due to heavy load). Although attractive, this solution would bring new problems, such as knowing when to create a new replica. Furthermore, the new replica has to be brought up to date, so this could lead at times to an overhead increase.

3.4 Example: MacroDB - Database Macro-component

So far we've presented the concept of the Macro-component in a general way that allows the use with almost any standalone software component. The presented architecture and the execution strategies discussed, can be used in different ways in order to take advantage of the design in a concrete implementation of the Macro-component.

In this section we will address the construction of a Macro-component specialized for databases. In particular, a Macro-component that replicates in-memory databases.

We focused on databases because databases are widely used in a variety of contexts. Furthermore, over the years multi-core processors started to emerge, but current database management systems (DBMSs) seem to struggle in taking advantage of these new architectures and do not scale properly [Sal+11].

Recently, in-memory database management systems (IMDBs) have become popular and are used in a large number of applications to manage internal data - e.g. Eclipse, OpenOffice - or as part of a storage system to provide better performance than simple traditional DBMSs [Kal+08; Cam+07]. As IMDBs do not incur in disk I/O overhead, it would be expected for these systems to scale with the number of cores. However,

tests [Soa+13] show that these systems have scalability issues with the increase of the number of cores, even in loads that do not conflict. Furthermore, the same tests reveal that it is not the lack of resources that causes this problems, but the design of current IMDBs.

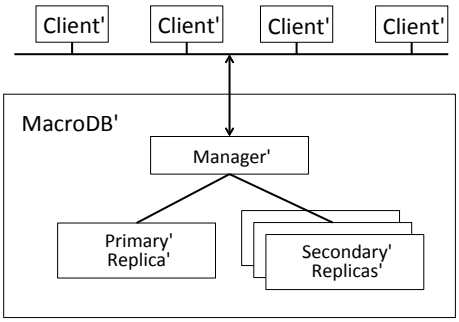


Figure 3.6: MacroDB architecture (Taken from [Soa+13])

MacroDB [Soa+13] is a Macro-component that aggregates a collection of coordinated IMDB replicas, for providing scalable database performance on multi-core systems. MacroDB works independently of the underlying database engine, acting as a transparent layer between applications and the database, thus making it easy to deploy with any database engine.

Figure 3.6 illustrates MacroDB's architecture. MacroDB is composed by two main components: *manager* and the *database replicas*.

- **Manager** The manager is responsible for coordinating transaction execution in the database replicas. Comparing this with Figure 3.2, we see that the manager and scheduler component were merged into only one component. But internally the job performed by the manager includes the scheduler's job.

- **Database Replicas** The database replicas are the engines where operations will execute and are divided in primary and secondary replicas. Each secondary replica has an associated executor that retrieves and executes pending updates that are waiting in a list of updates.

Clients remain oblivious of the replicated nature of the database engines. That is because, the manager offers a *JDBC* compliant interface, in which clients invoke their operations. From the client's point of view, it seems like the client is sending operations to only one concrete database engine. Besides providing an homogeneous interface, the manager forwards client's requests to the appropriate replica(s) on reception. Dividing the replicas in primary and secondary has its purpose and we will briefly explain them in the next section.

3.4.1 Transaction Execution

MacroDB uses a master-slave replication scheme [Wie+00b]. The master maintains a primary copy of the database data, while the slaves maintain secondary replicas. Consequently, update transactions are concurrently executed on the primary replica, being asynchronously propagated to the secondary replicas. Each secondary replica stores a version counter that keeps tracks of the number of commits on that replica. We now explain how update and read transaction are processed in more detail.

Update Transactions To identify a given transaction as being an update transaction (or not), it is required to invoke the setReadOnly method from the *JDBC* interface with the associated value (true or false).

When an update transaction starts, the runtime associates a newly created batch with that transaction. All operations are executed in the primary replica and if the operation is an update, it is inserted into the associated batch. On a successful commit, the version number associated with the primary is incremented. Furthermore, that version number is stamped in the batch, and inserted in the list of pending batches for the secondary replicas. To guarantee correct transaction ordering, it is necessary to make sure that no other transaction starts and ends before a transaction that has already started a commit, thus this operation requires synchronization with other threads. When a rollback occurs, either by user request or error, the rollback is only executed in the primary replica and the associated batch discarded.

To deal with pending batches, each secondary replica has an executor that polls the list, waiting for the next batch to be inserted and executes it. These batches are executed sequentially, thus avoiding possible deadlocks. Atomically the executor will commit the transaction and increase the associated version number in the secondary replica. By executing batches in the same order as the commit in the primary replica, the secondary replicas are guaranteed to evolve to the same consistent state.

Read-only Transactions When a transaction is identified as read-only, it is executed directly on a secondary replica in the context of the caller thread (to minimize overhead). MacroDB enforces that such transaction will only execute on a up-to-date replica, i.e, when its version is at least equal to the version number in the primary replica. To do that, when a read-only transaction starts and prior to the execution in the secondary replica, the manager reads the version number on the primary replica. This number defines when the secondary replica executes the transaction, waiting if necessary until the secondary replica has the correct version, i.e, until the version number is at least equal to the primary's version number. This guarantees that the transaction does not execute in an out-of-date replica. With this, MacroDB provides clients with a *single copy serializable view* of the replicated database [Soa+13]. This strategy is very similar to the read-one strategy presented in Section 3.2.2.

It should be noted, that by executing update and read-only transactions in different replicas, update transactions never block due to concurrent reads. Furthermore, contention is reduced because transactions are distributed across the replicas.

3.4.2 Results

Having explained how MacroDB works, we will now show how it performs. The basic idea behind the design is to distribute load across replicas.

The benchmark used to test the system was TPC-C, with four different workloads: standard (8% reads and 92% writes), 50-50 (50% reads and 50% writes), 80-20 (80% reads and 20% writes) and finally 100-0 (100% reads and 0% writes). For non-standard workload, for each type of transactions (read-only or update), we have kept the same ratio as in the original workload - e.g. for a read-only workload, the same number of transactions is executed for the two type of read-only transactions defined in the benchmark. We'll designate the first two as write dominated workloads and the remaining ones as read dominated workloads. The engines used to perform the benchmark were H2[1] and HSQL[2] and two MacroDB configurations using the same engines (MacroH2 and MacroHSQL). The tests were performed in a SunFire X4600 M2 x86_64 server machine, with eight dual-core AMD Opteron Model 8220 processors and 32GByte of RAM, running Debian 5.

Write dominated workloads In write dominated workloads the MacroDB had limited scalability. In fact it showed a little overhead sin the standard workload. In this workload, the standalone engines managed to outperform the MacroDB configurations. This puts in evidence the overhead of the MacroDB, which ranged between 5% and 14%, compared to standalone engines. However in a moderate write workload, namely the 50-50, the MacroDB configuration managed to offer between 40% and 70% performance improvement (HSQL and H2 respectively). The stress at the primary replica is still noticeable, as an increase in the number of replicas in this workload, did not result in an performance increase [Soa+13].

Read dominated workloads The MacroDB design should improve performance more noticeably in read dominated workloads. This is because with an higher read rate, the MacroDB is able to distribute more work across replicas. This is verified by the results, where both MacroDB versions achieved improvements ranging between 93% and 176%, with the number of replicas varying between one and four. H2 was the engine in which the gain in performance was more noticeable. The same benchmarks were run, with six replicas, in order to determine if the resources were being fully utilized. The results showed that were still resources available as there were performances improvements up to 234%. These results seem to indicate that as long as the machine has resources to use,

[1] http://www.h2database.com
[2] http://hsqldb.org/

MacroDB is able to take advantage of existing resources assuming that the workload is read dominated.

Memory usage As stated in Section 3.3, if software components did not share data between them, this would result in an almost linear increase of memory use when using Macro-components. In the case of the IMDBs it seems that IMDB replicas share immutable Java objects such as Strings. For that reason, even when running a MacroDB setup with four replicas, memory consumption increase ranged between 1.7x and 2.5x (H2 and HSQL respectively) compared to the standalone engines. Thus deployment of a MacroDB setup with several replicas is possible. Furthermore, modern machines have large amounts of memory and this growth shows no sign of stopping, making such setups feasible.

4

Distributed Macro-components

This chapter presents the design and implementation of distributed Macro-components, which extends Macro-components for a set of machines running in a cluster.

Section 4.1 discusses the rationale for distributed Macro-components and replication alternatives. We proceed by presenting the solution's architecture in 4.2, detailing the inner workings of each component.

Section 4.3 presents distributed MacroDB, a distributed version of MacroDB presented in Section 3.4.

4.1 Introduction

A large number of different services rely on replication in multiple machines to provide fault tolerance and scalability. In this work, we study how to combine replication across machines with replication within a machine. This combination has the potential to provide scalability and fault-tolerance by replicating a service in multiple machines; and to fully explore multi-core machines by relying on replication inside each machine. To implement this vision, we will introduce distributed Macro-components, an abstraction that extends Macro-components to multiple machines. Our basic design relies on replicating Macro-components across multiple machines. We name an instance of a Macro-component a network node (for avoiding confusion with replica, which we continue using for a replica inside a Macro-component).

The main goal of this work, is to build a system that works correctly and is able to demonstrate its potential. In our work we make the following assumptions.

Replicas are deterministic As mentioned in the previous chapter, replicas must have a consistent state. For this reason, different replicas are assumed to be functionally equivalent and deterministic. As consequence, for any sequence of operations executed in a replica, the results of executing the same sequence of operations in another replica from the same initial state will be the same. This assumption guarantees that operation executed in any of the replicas will have the same result and that the internal state of each replica will not diverge when executing the same sequence of operations.

Replicas do not fail arbitrarily To fully guarantee replica consistency determinism alone is not enough. Arbitrary failures (e.g, implementation bugs) may cause the Macro-component to obtain conflicting results from different replicas, or somehow corrupt their internal state. For the purposes of this work, we assume that replicas do not fail arbitrarily.

Crash model The two previous points specify requirements applicable to the Macro-component replicas. For the network nodes and the network itself some assumptions are also made. We assume that network nodes do not fail in a byzantine way. But they can fail by crash, i.e, a network node may halt prematurely, but until it halts the node behaves correctly. Once it halts it never recovers [Bud+93; Lam+84]. The remaining network nodes seamlessly deal with a failure of another node, i.e, they ignore the failed network node.

Network partitions In the network itself we assume that no partitions are formed during normal execution of the system. The existence of partitions is not a limiting factor by itself. The problem arises when the network partition is fixed and naturally the network nodes in each partition need to recover (or merge) their state [JP+02]. Such mechanism does not exist.

Message delivery We assume disseminated messages eventually are delivered at the destination. Packets may be dropped or delayed, but are guaranteed to be delivered. This way there is no need to make assumptions in the maximum time needed to wait for a message. With this assumption application code may simply wait until a message arrives.

4.1.1 Replication Strategy

In this section we discuss two important design decision that we need to take when designing distributed Macro-components: whether all replica replicate all data; and how to design the replication protocol to guarantee that all replicas are updated while minimizing the protocol overhead.

4.1.1.1 Full Replication vs. Partial Replication

When building a distributed system, this question often needs to be evaluated according to the system's objectives and needs. As mentioned before, replication is a mechanism to provide fault-tolerance and to improve performance. The challenge usually is in keeping all network nodes consistent.

With full replication, every network node has an entire copy of the dataset. The downside of full replication is that all network nodes need to keep their dataset coherent when an update occurs in any network node. In other words, all network nodes need to be aware of all updates. This may impose a significant amount of overhead, depending on the number of network nodes deployed. But having all data replicated across all network nodes has its advantages. For instance, it is easier to distribute work among replicas, and by minimizing the work done by each network node, throughput is likely to increase. Many of the techniques used with full replication were previously discussed.

With partial replication the system's dataset is not fully replicated in each network node. Instead, each replica only stores a subset of the dataset. With this scheme a system is able to deal with a replica that cannot hold the entire dataset contrary to the full replication scheme, being this, one of its advantages. With partial replication not all network nodes need to receive updates of a given operation, only the network nodes that contain the associated data. This helps reducing the network overhead imposed by full replication, and in theory allows for more network nodes to be deployed without harnessing scalability [Sch+10; Pel+12] while still providing a decent level of fault-tolerance possibilities. Furthermore, full replication naturally increases access locality but as mentioned, at the cost of exchanging messages with all network nodes. With partial replication to execute a given request it may be necessary to contact more than one network node. This may increase the protocol's complexity and thus create overhead sources. In the scope of this work, although both could be used, to the additional complexity of partial replication in database systems, we focused on full replication.

4.1.1.2 Macro-Component Replicas

Regarding the Macro-component replicas, there are at least two options: treat each Macro-component replica as a individual node of the system, or have some kind of a primary replica (or any similar scheme) that receives the updates, and later propagates them to the remaining replicas in a multi-level approach.

Treating each Macro-component replica as an individual entity is a straightforward solution. The only requirement is to connect each replica to the communication system (explained in 4.2.1) and each individual replica will use the system as necessary for propagating updates among other tasks. A characteristic of this approach is its simplicity. On the other hand one, can imagine an example where each Macro-component has for example, three replicas. It is only necessary to have two network nodes for the number of replicas to be involved in the replication protocol to be very high. In this case we'll

have six replicas in total. Due to this, the replication protocol potentially becomes very expensive, significantly harnessing the system's scalability.

Having a replica (or a set of replicas) responsible for receiving the updates and later propagating them to the remaining nodes may be more advantageous. On one hand it is not so straightforward as it is necessary for the Macro-component to have a mechanism that allows the execution at a given replica to coordinate with replicas on remote Macro-components before propagating the result to other Macro-component replicas. But using such scheme, although more complex, does not require that all replicas participate in the replication protocol.

In the context of this work, we assume that the underlying Macro-component presents a primary-secondary scheme. The reason for this is because in distributed systems usually the most expensive operation is network communication. By forcing the primary-secondary scheme in the Macro-component it helps reducing that cost by having only one of the replicas participating in network coordination.

4.2 Proposed Architecture

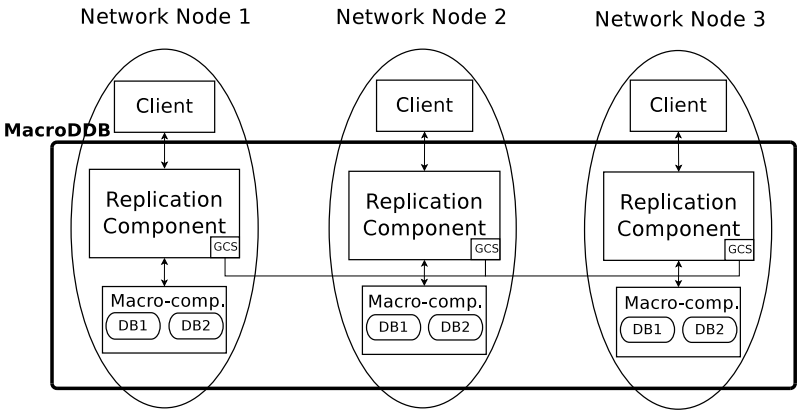


Figure 4.1: MacroDDB Deployment Architecture

In Figure 4.1, we present the proposed distributed Macro-Component (DMC) architecture. The design is intended to be as general as possible in order to support any Macro-component.

The DMC is formed by the client, the replication component and by a Macro-component. A network node is composed by the replication component and the Macro-component. An arbitrary number of clients may connect to a given network node. The clients interact with the system by using an interface with the same operations that the replicas provide. The clients are either local (have a direct reference to the replication component) or remote (communicate with the replication component through the network).

The replication component works as a *middleware* between the client and the Macro-component. When the replication component receives a request, it redirects it to the Macro-component. As the interface that the client uses is identical to the Macro-component's

interface, redirection becomes mostly a one to one relation, making the replication component work as a proxy. The implementation of the replication component will have to deal with the replication requirements and other necessary operations (e.g, consistency requirements). At each replication component, there is a request queue in order to receive update propagation from the remaining network nodes or operation requests from remote clients.

4.2.1 Communication System

The communication system is the component that provides the means for the clients to communicate with the network nodes (when applicable) and for network nodes to communicate with each other.

For inter-network communication, we used a Group Communication System (GCS). The GCS used is the same as the one presented in [Val12] and the most relevant interface operations are shown in 4.1

Table 4.1: Group Communication System interface

Function	Description
TOBCAST(M)	Broadcasts using total order the message M
RUCAST(M, D)	Reliable unicast message M to destination D. Used for point to point communication
GETADDR()	Retrieves the local address of the node

The TOBCAST primitive sends messages to all nodes in the group and guarantees that all nodes reliably deliver the messages in the same order. This operation is used, for example to propagate operations to all nodes. RUcast reliably propagates a message to a single node. The RUCAST may be used by remote clients to direct requests to a specific remote node. Furthermore, requests replies will also use the unicast primitive (directed to the corresponding client) as there is no need to broadcast the replies to every client and it would only cause overhead.

Using a communication system that provide generic operations, allows to abstract the users from the concrete implementations. Three implementations were considered: JGroups [Ban98], Appia [Pin01] and Spread [Ami+00]. From the three we choose to use JGroups, as it provides the highest throughput [Val12].

4.2.2 Client

The clients use the same interface as presented in the Macro-component but the implementation has to deal with the fact that the target component of an invocation may not be local but rather remote. The client has access to the message sender subcomponent that abstracts the client from the inherent complexities of communication. The provided interface by the message sender is presented in Table 4.2.

Table 4.2: Message sender's interface

Function	Description
RUNOP(MO)	Synchronously sends the macro-operation MO to the network nodes, waiting for the associated reply. It uses a TO-broadcast primitive
RUNOPRELIABLY(MO, DA)	Reliably and synchronously sends the macro-operation MO to a destination with address DA.
RUNOPASYNC(MO)	Asynchronously sends the macro-operation MO to the network nodes, without waiting for the associated reply. It uses a TO-broadcast primitive

The RUNOP operation act as a normal method invocation. This function will broadcast the operation to all reachable network nodes. The RUNOPRELIABLY does the same job as the previous operation, but instead of relying on a TO-broadcast primitive, it directs the message to a specific network node. It is useful when a client associates itself with a particular network node, and does not need to broadcast its operations to every node.

The RUNOPASYNC operation contrary to the previous operations it does not wait for a reply, it immediately returns a result that allows retrieving the final result sometime later in the future. Such invocation semantics may enable some optimizations, e.g., the use of speculation [Wes+09; Kot+07] which is a technique that tries to predict results in order to improve performance.

The responses provided by the remote nodes, are received by the client's message listener later explained.

Macro-operation A macro-operation encodes a method invocation, including the method to execute and its parameters, i.e, transforms the client request into an invocation that is recognized by the underlying Macro-component. For each operation in the Macro-component interface it is required to exist a macro-operation. For each macro-operation, a request ID is obtained and inserted into the macro-operation. This way, the replication component is able to insert into the reply that ID, that allows the client to identify the request for which it is receiving a response. The macro-operation has two functions in its interface as presented in Table 4.3.

Table 4.3: Macro-operation's interface

Function	Description
EXECUTE(MC)	Uses the functions of the Macro-component MC to implement the specification.
EXECUTE(MC, MT)	Uses the functions of the Macro-component MC to implement the specification and may use the Message Translator MT to create a new Macro-component server side.

In each macro-operation's function, the operations provided by the associated Macro-component are used to implement the specification. The second function on Table 4.3 is a helper function so operations are able do dynamically create new Macro-components. This function is usually used when there is some kind of hierarchy on the Macro-component. For instance, MacroDB presented in Section 3.4 is one such example. The Message translator (MT) referenced in the second operation is a replication component's subcomponent and it will be further explained in the next section.

Client Message Listener The client message listener is the client's component that listens for incoming messages. It is specially useful at receiving back the responses sent by the replication component. In our case all other type of messages are discarded by the message listener, i.e, if it is not a response type message it will be discarded.

4.2.3 Replication Component

The Replication component (RC) is responsible for dealing with clients and other network nodes requests. Furthermore, this component is the only one that deals with the replication of data/operations to the remaining network nodes in the system.

The RC is composed by several subcomponents in order to separate responsibilities across these subcomponents. The mentioned subcomponents are depicted in Figure 4.2 with the arrows illustrating the direction of the usual execution flow. A more detailed explanation of these components is given below in a bottom-up approach.

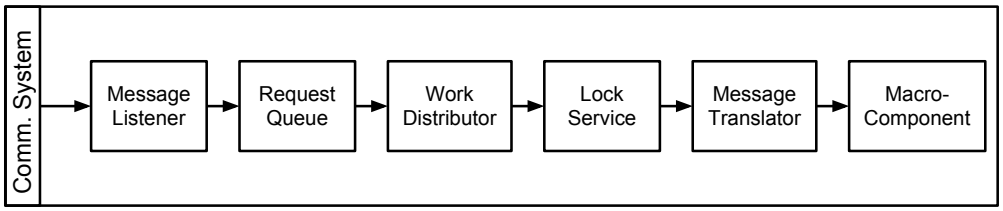


Figure 4.2: Replication component's structure

4.2.3.1 Message Listener

The Message Listener subcomponent is responsible for receiving incoming messages from remote clients and other network nodes. It is used as the upcall of the communication system. Upon reception, the message listener pre-processes the request and inserts it in the request queue using the provided interface, where it will remain until it is chosen to be processed. By delegating request processing into another component, we are able to immediately release the communication system resources and receive other messages from the same or remaining network nodes and clients. Furthermore, it avoids having the receiving thread blocking which may result in a crash suspicion.

The message listener has only two pre-processing tasks: reply to probes and encapsulate the content into a more friendly representation before reaching higher levels.

The message listener automatically replies to discovery messages, used for the different components to connect to each other.

The other pre-process task is to encapsulate the content received. The upcalled method has the following header:

Listing 4.1: Message Listener receive header

```
1 onDelivery(Object obj, Address src, int size)
```

The `obj` parameter is simply a sequence of bytes which is left untouched to avoid wasting time. What is done is to insert the content and address (which is later necessary for replying) into a new object which offers methods to easily access the contained information. Finally the message is passed to the upper level.

4.2.3.2 Request Queue

The Request Queue is the container for the received requests. It interfaces with the message listener and work distributor (insert and retrieve respectively). The internal structure used is application dependent. For instance, one can use a simple FIFO structure or a priority aware structure when requests have different priorities. In this case a priority structure is more natural, and it only requires for the requests to have some kind of priority identifier (e.g, a number) to define their priority.

The structure used in this component should be ready to deal with multiple threads accessing it, as many concurrent threads may be inserting and removing at the same time. The request queue component provides the interface shown in Table 4.4

Table 4.4: Request Queue's interface

Function	Description
INSERTREQUEST(RQ)	Inserts the request RQ into the internal structure used for the request queue.
GETREQUEST()	Retrieves the next pending request from the structure defined by the internal implementation.

The `insertRequest` function is used by the message listener while the `getRequest` is used by the work distributor. In our solution these are the only components that use the interface provided by the request queue.

4.2.3.3 Lock Service

The Lock Service is an optional component that may be used by the work distributor. It controls when and how many request jobs may start executing. In its basic form it provides control following a single writer multiple readers semantics also known as *read-write* lock.

A read-write lock resembles a mutex, in that it controls access to a shared resource. It allows concurrent access to multiple threads for reading but does not allow access to more than one writer to update a resource. For instance, when one wants to avoid a reader of reading different values from the same resource even when no update operation has been performed from that thread, one may wish to block the reader until all precedent updates are completed. In that case this basic form of control is optimal and allows a consistent view of the resource state at all times.

An illustration of the implementation code is shown in Figure 4.3 which also illustrates the provided interface. When a thread wants to start executing a request it invokes the BARRIER function on the lock service. The lock service has a lock and an associated condition variable. If it is an update then it must test if any other thread is reading or writing (Line 8) and if so, it waits on the condition variable. Otherwise it affects the control boolean value to true (Line 11) and releases the lock so other threads can continue if there were any other threads blocked at method invocation. When the thread in question finishes writing it invokes the FINISHWRITE method putting the boolean control value to false, and signaling (Line 31) the change, so other threads blocked on the condition variable are able to proceed.

If it is not a write operation (i.e, is a reading operation) then the only condition that should be tested is if any writer thread is executing (Line 13). Recall that this lock service allows multiple readers. If any writer thread is executing then the calling thread must wait on the condition variable, else it proceeds increasing the number of readers by one. When a reader finishes executing it decrements the readers counter and signals (Line 24) the remaining blocked threads (only writers should be blocked), if no other reader is currently executing.

```
1: lock ← CREATELOCK( )
2: condition ← CONDITION(lock)
3: isWriting ← false
4: readCount ← 0

5: function BARRIER(isupdate)
6:     OBTAINLOCK(lock)
7:     if isupdate then
8:         if isWriting or readCount > 0 then
9:             WAIT(condition)
10:        end if
11:        isWriting ← true
12:    else
13:        if isWriting then
14:            WAIT(condition)
15:        end if
16:        readCount ← readCount + 1
17:    end if
18:    RELEASE(lock)
19: end function

20: function FINISHREAD
21:     OBTAINLOCK(lock)
22:     readCount ← readCount − 1
23:     if readCount = 0 then
24:         SIGNAL(condition)
25:     end if
26:     RELEASE(lock)
27: end function

28: procedure FINISHWRITE
29:     OBTAINLOCK(lock)
30:     isWriting ← false
31:     SIGNAL(condition)
32:     RELEASE(lock)
33: end procedure
```

Figure 4.3: Lock Service pseudo-code

4.2.3.4 Work Distributor

The work distributor component removes the pending requests from the request queue, and puts them to execution. When a request is retrieved from the queue it passes through the lock service to determine if the request is delayed or put immediately to execution.

The work distributor resorts to a thread pool to execute the requests. The number of threads initially created are defined by the user. Otherwise this component uses the number of available cores in the physical machine to build the thread pool. Consequently the number of threads allowed to execute concurrently determines the number of clients that are able to execute their requests in true parallelism. The threads are all created at initiation time, thus avoiding the costs of dynamic thread allocation. When the number of threads is exhausted the task job is simply inserted into a wait queue, and as soon as there are resources for the job it is put to execution.

Until now, the messages received by the replication component are passed to the upper levels without change, i.e, as a set of bytes. Recall that we choose this way in order to release the communication system resources as soon as possible. The only task done so far was to encapsulate it in a more friendly manner to access certain attributes. At this stage, the set of bytes is transformed into a higher level representation before executing its code. In this case we are referring to the reconstruction of the content into a *macro-operation*. Afterwards, this macro-operation is supplied to a thread (of the thread pool) which will effectively execute the operation using the message translator explained below.

4.2.3.5 Message Translator

The Message Translator (MT) subcomponent is the core piece of the replication component. The thread mentioned in the previous subsection will use the MT to effectively access the underlying Macro-component and execute the macro-operation. The MT presents the interface shown in Table 4.5 and is used by the invoking thread.

Table 4.5: Message Translator's operation execution interface

Function	Description
MAKECALL(MO)	Executes the macro-operation MO in the associated Macro-component issued by a remote node.
MAKECALLLOCAL(MO)	Executes the macro-operation MO in the associated Macro-component issued by a client (remote or local).

It is possible for several Macro-components to exist which are jointly managed in this subcomponent, thus all of the Macro-components are stored in the MT. The received macro-operation should bring with it the identifier of the Macro-component for which it wants to access. Having the identifier, the MT is able to fetch the Macro-component and use it as argument to one of the functions presented by the macro-operation's interface

as shown in Table 4.3. Afterwards the macro-operation will simply use the interface provided by the Macro-component to implement the desired behaviour.

After execution, a result is returned to the client. This way handling requests in the remote client side is simplified as the pattern is the same for all requests: send, wait for response and process response. One question that arises is, how to discover which (remote) client to reply to. To solve this, when the execution thread is created, the sender's address is also provided. When the method execution completes the communication system is used to reliably delivers the response to the client. In this case, total order is not necessary as the client is only communicating with one network node, and that network node will only send one reply at a time. The only thing that has to be guaranteed is that the response reaches its destination, and that is taken care by the communication system. If it is a local client, no communication is necessary and the result is simply forwarded to the client.

4.3 MacroDDB: Replicated and Distributed Databases

Having described the general design of a DMC, we will now focus on the design and implementation of a concrete example. In this case, we will focus on databases due its wide usage and abundant research on how to distribute them [Cor+12; Eln+06; Har+08; Eln+05].

4.3.1 Overview

In Chapter 3 we presented results that demonstrated the lack of scalability of database systems in multi-core machines, thus limiting throughput. This limitation results in scalability problems of the replicated database systems that are frequently used, for instance, to support several web services [Sal+06]. To deal with this problem, we present the design and implementation of the MacroDDB system, a solution based on hierarchic database replication that supports *snapshot isolation* semantics. The database is replicated across several network nodes, in which each node keeps a set of fully replicated database replicas.

MacroDDB has an organization in which each network node has one replica that is denominated master (or primary replica) and a variable number of secondary replicas. A read-only transaction executes exclusively on a secondary replica, thus no messages are exchanged between replicas of the same network node or replicas from different network nodes. Update (or write) transactions execute initially on the primary replica of the network node in which the client initiated the transaction, until the moment that the transaction must be validated. At that moment, the transaction is validated against (remote) concurrent transactions, verifying the existence of write conflicts with the transaction that is committing. This technique works by using a group communication system as proposed by Kemme *et al.* [Kem+00]. After transaction validation the remaining local

replicas are asynchronously updated.

The hierarchic approach minimizes the number of exchanged messages in the network and the number of replicas in which the transaction validation needs to be performed. Furthermore, by executing the transactions in the secondary replicas in batch (Section 3.4.1) the execution time is reduced and consequently, the replicas load is also reduced.

4.3.2 Snapshot Isolation

Snapshot isolation (SI) is a multiversion concurrency control mechanism used in databases (and in STM). An attractive property of SI is the fact that readers are never blocked or aborted, and readers never cause writers to block or abort. This advantage is significant for workloads with a large fraction of read-only transactions (such as those resulting from the generation of dynamic Web content) [Eln+05].

In this work we will use the following definition of SI [Pla+04]:

SI A transaction T_i that is executed under snapshot isolation gets assigned a start timestamp $begin(T_i)$ which reflects the starting time. This timestamp is used to define a snapshot S_i for transaction T_i. The snapshot S_i consists of the latest committed values of all objects of the database replica where T_i executes initially at the time $begin(T_i)$. Every read operation is issued by transaction T_i on a database object x reads the version of x which is included in the snapshot S_i. Updated values by write operations of T_i are also integrated into the snapshot S_i, so that they can be read again if the transaction accesses updated data. The set of write operations of T_i is the write set of T_i (WS_i).When transaction T_i tries to commit, it gets assigned a commit timestamp $end(T_i)$, which has to be larger than any other existing start timestamp or commit timestamp. Transaction T_i can only successfully commit if there exists no other committed transaction T_k having a commit timestamp $end(T_k)$ in the interval $\{begin(T_i), end(T_i)\}$ and $WS_k \cap WS_i \neq \{\}$. If such a committed transaction T_k exists, then T_i has to be aborted (this is called the *first-commiter-wins* rule [Gra+96], which is used to prevent lost updates). If no such transaction exists, then T_i can commit (WS_i gets applied to the database).

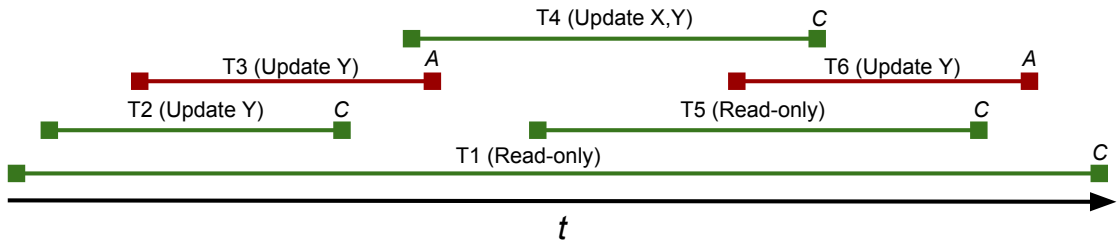


Figure 4.4: Snapshot Isolation execution example

A sample execution of transaction running in SI mode is given in Figure 4.4. The squares at the edges of the lines represent the *begin* and *end* of a transaction respectively. The lines colored green (or with the letter C in the end) mean that the transaction successfully committed. On the other hand the red colored lines (or with the letter A in the end) mean that the transaction had to be aborted. Transaction T_1 is a long running transaction, but since it is read-only, i.e, its write set is empty ($WS_1 = \{\}$), it will never be blocked by other transactions, nor will it block other transactions. Concurrent updates (like T_2, T_3, T_4 and T_6) are hidden from T_1. T_2 updates item Y. As there is no other concurrent transaction that conflicts, T_2 is able to commit. T_3 however, concurrently updates item Y and thus the intersection of T_2 and T_3's write set is not empty: $WS_2 \cap WS_3 \neq \{\}$. Due to the fact that T_2 committed first and the *first-committer-win* rule, the transaction manager is forced to abort T_3. T_4 write set does intersect T_3 write set but due to the fact that T_3 was aborted the conflict is not considered, and thus T_4 is able to commit. In T_6, the write set intersects with T_4's write set. And again by the *first-committer-win* rule T_6 must be aborted. Finally, transaction T_5 is read-only and falls in a case identical to T_1 where the transaction can not be blocked or aborted, nor blocks or aborts other transactions.

4.3.3 Deployment Architecture

Figure 4.1 illustrates MacroDDB's deployment architecture when using a three network node configuration. In each node the presented configuration is similar to the general design presented in the previous section. The replication component has been slightly augmented in this context. In MacroDDB we have a request queue that is used exclusively for remote clients and another used to receive update propagation from the remaining network nodes. For each request queue there is a work distributor associated. This way we achieve a separation that allows update propagation and remote clients requests to be processed concurrently. In order to make the network nodes evolve through the same intermediates states and consequently final state, we must apply the propagated updates in the network node respecting the order that resulted from the total order broadcast. To do that, the work distributor that deals with update propagation only uses one thread. This way the updates are serialized and forced to comply the order given by the total order broadcast.

Relaxing the order constraints in multi-core systems has already been addressed in [Kap+12], but it relies on optimistic execution techniques and may incur in additional rollback overhead (and the mixer component overhead). To keep the prototype simple we decided to serialize the updates at reception.

In what concerns the clients, as the order of updates are only decided at commit time, clients are allowed to execute their requests concurrently. Consequently the work distributor associated with the clients's queue is initiated with the remaining available resources, i.e, the remaining processor cores available.

4.3.4 Transaction Execution

So far we've presented the inner components that form the MacroDDB system. In this section we will detail how transaction execution is performed. To invoke an operation on the MacroDDB, the client's application connects to the replication component of a particular network node. The replication component defines the logic of operation processing.

For each initiated transaction, all associated operations are initially executed in the local MacroDB (which is logically under the replication component) using the JDBC interface. Each individual operation may execute in the MacroDB's primary replica, if the transaction is an update transaction (i.e, contains update operations), or in one of the secondary replicas, if the transaction is signaled as a read-only transaction (as explained in 3.4.1).

Transaction processing at commit time differs. With read-only transactions the commit operation is invoked and the transaction successfully terminates (assuming that the DBMS guarantees that a read-only transaction never fails at commit).

When dealing with write/update transactions, validation becomes necessary. To do that, potential write-write conflicts with transactions that executed concurrently are verified (as specified by the snapshot isolation semantic explained in 4.3.2). This verification is done by using a total order multicast primitive to propagate the write set formed as the client issues write operations.

This approach, previously used in several other systems [Kem+98; Eln+05], guarantees that the propagated write sets are delivered and processed in the same relative order in every network node. In each network node, the received write sets (and associated transactions) are tested against any concurrent transaction that terminated after the start of the transaction that is being verified. The start of a transaction is marked as the number of update transactions that the network node has completed so far. Since all network nodes perform verification in the same order, then all network nodes will reach the same final result and commit a transaction if no conflict is found, and abort otherwise. Every network node's replication component stores the history of commited (and aborted) transactions, where the verification algorithm will fetch the necessary transactions for comparison. In the remote network nodes, i.e, all the nodes except the node where the transaction started, the commit operation is preceded with the application of the received write set in the local MacroDB's primary replica. The network node that received the client's transaction operations returns the commit result to the client.

4.3.4.1 Implementation

Some implementation details were omitted for clarity in the previous description. In this section we will detail some aspects of transaction execution.

Verification For the correct execution of the proposed solution it is required to extract each transaction's write set and avoid possible deadlocks from the interaction of remote

transactions with local transactions. To implement these functionalities known solutions from other systems are used. In particular the use of SQL statement analysis using JSqlParser [Man+13] to extract the write set and the analysis of possible deadlocks before executing the operations of a remote commit. If such conflict exists, the local transaction is aborted.

Aborting the local transaction makes sense for three reasons: first, aborting a remote transaction would incur on additional overhead, because the local running transactions potentially are not the same at every network node, as such the information that the remote transaction was aborted would need to be broadcast to every network node; second, if a local transaction is in conflict with a remote that is being verified at commit time, then in the transactions history the remote transaction terminates after the start of the local transaction. Thus, according to the snapshot isolation definition that we used, the local transaction would abort anyways if by any chance its write set got broadcasted; and third the transaction has already executed its operation and is ready to terminate, aborting it would result in wasted resources.

Client identification As stated before, the interface shown to the user is the JDBC interface. As such, each client has an associated `Connection`, from which it is able to create `Statement` and `Prepared Statements`. Each connection retains the write sets formed by the transactions executed in that connection. The replication component needs to associate each connection with a client (local or remote). To do so, when a client asks for a connection to the database, it sends a macro-operation to the replication component. The macro-operation is then responsible for creating an unique ID that associates that connection with the client. Each connection is stored in a data structure that maps IDs to connections. More specifically, it associates IDs to a created objected that will encapsulate all connection's information of that client, that includes `Statements` and `Prepared Statements`. In the end, the created ID is sent back to the client. From hereafter, all operations issued by the client must be identified with that same ID. This way the replication component is able to determine which connection should be used. We do require that different clients have different connections.

Macro-Operation In Section 4.2.2, an overview of the macro-operation was given. We introduced the interface provided by the macro-operation and the goal of defined operations. What was left out, was the advantages of having the operations implementations in such abstraction and an example to show how they are used.

An advantage of having the implementation of the operations concentrated in one function is that basically most of the logic to be implemented in a specific context is on these macro-operations. As consequence, in the presence of bugs they will likely be concentrated in these macro-operations and so it becomes easier to find, detect and correct bugs. Recall that the implementation that we refer, is the transformation from a remote request into the invocation of an operation recognized by the Macro-component. Another

advantage is the fact that is possible to perform arbitrary computation in the macro-operation. For instance, consider that a given result from the underlying MacroDB is not serializable (and it does happen), and thus unable to send the content through the network. With macro-operations it is possible to process the results and make it proper for network traffic. On the other hand, this approach may induce some overhead, and it is arguable if the ability to perform arbitrary computation may lead to the creation of complex code or *spaguetti* code for operations that require higher number of code lines. But most of the operations in the context of the MacroDDB are used as a proxy to the underlying MacroDB. As an example we present a macro-operation in Listing 4.2. In this case, it reflects the commit operation. Error handling is omitted.

Listing 4.2: Macro-operation of the Commit operation

```
1 public Object execute(MacroConnection mc)
2 {
3     Connection c = mc;
4     c.commit();
5
6     return new EmptyResponse(getID());
7 }
```

Before invocation, the ID that identified the client's connection was used to obtain the associated connection, and passed as argument to the EXECUTE function. The macro-operation may return three types of objects: an EMPTYRESPONSE corresponding to void operations; a RESPONSE which receives the response content for non-void operations; and an EXCEPTIONRESPONSE when an operation terminates in error. By differentiating the types of responses the client's application is able to react accordingly.

5

Evaluation

This Chapter reports the results of an experimental study aiming at evaluating the performance and the impact of our solution for replicated in-memory database. The evaluation was performed using a standard benchmark.

Section 5.1 starts by emphasizing some important aspects about our implementation of the proposed model. Next, in Section 5.2 the benchmark used is explained in order to help understand it better, while presenting an adapted distributed version. We proceed by explaining the experimental setup/settings for the experiments in Section 5.3. We then evaluate the overhead introduced by our solution in 5.4. We proceed with a comparison of our approach (multi-master with primary-secondary) with the standalone multi-master case in 5.5, followed by some scalability results in 5.6.

5.1 Implementation Considerations

We would like to note that the replication component presented in the previous chapter is a generic component. It was not hard to adapt it to the MacroDB system, the changes required were the interface shown to the user, which varies with the underlying component, and some specific changes that are related with the context such as storing transaction history, the association of each connection with the individual clients and conflict verification. These additions do not change how the component invokes operations on the Macro-component and only the macro-operations specific to the replicated databases case make use of them. As previously stated, each of the operations provided to the client has a concrete implementation (different from what is provided by the standard component). The modified operations work mostly as a proxy and could be easily auto-generated. Previously, in a preliminary stage of this work a Macro-component targeting

java data structures was used to test the replication component. The changes necessary were made on the macro-operations and not in the component itself as at the time we sent every operation using the TO-broadcast primitive. Furthermore, the use of macro-operations and the interface provided by the replication component's subcomponents, easily allows the use of any communication protocol.

5.2 Benchmark

To evaluate our system we have used a known standard benchmark: TPC-C.

The TPC benchmark C (TPC-C) is an on-line transaction processing (OLTP) benchmark from the Transaction Processing Council [Tra13]. TPC-C defines a complete environment where a set of terminal operators executes transactions against a database. The benchmark defines its operation around the principal activities (or transactions) of an order-entry environment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses [Tra13]. TPC-C involves a mix of these five transactions that are of different types and complexity. This way different components of the tested system are being stressed by having multiple transactions of different natures competing for system resources. In summary, the TPC-C benchmark tries to represent any industry that must manage, sell, or distribute a product or service.

The benchmark specifies 9 tables, with the relations illustrated in Figure 5.1

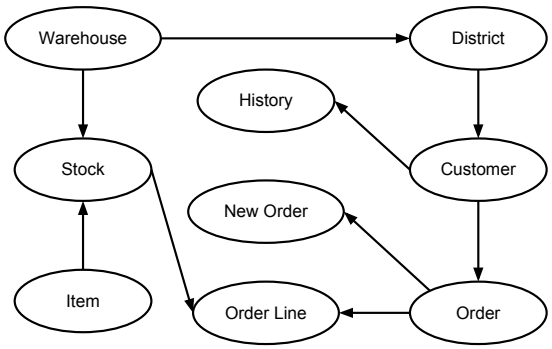


Figure 5.1: TPC-C table relation (Adapted from [Tra13])

The arrow defines foreign keys on the tables. For instance, Warehouse table points to Stock table, meaning that the Stock table has a foreign key referencing the Warehouse table. In other words a pointed table has a foreign key referencing the table that is pointing it.

The TPC-C standard benchmark workload includes 92% of write transactions and 8% of read transactions. We have used other mixes of read and write transactions for experimenting different levels of read-only transactions: 50-50 (50% reads and 50% writes),

80-20 (80% reads and 20% writes) and 100-0 (100% reads) or read only. In these workloads, we have increased/decreased proportionally the percentage of each transaction type.

The default dataset used is composed by: 2 wharehouses, 10.000 items (per wharehouse), 10 districts (per wharehouse) and 300 customers (per district)

5.2.1 Transactions

TPC-C benchmark is composed by the: `New-Order`, `Payment`, `Order-Status`, `Delivery` and `Stock-Level` transactions. The transactions are briefly described below [Tra13].

- **New-Order** The new-order transaction consists of entering a complete order through a single database transaction. It consists in read-write operations.
- **Payment** The payment transaction updates the customer's balance and reflects the payment on the district and warehouse sales statistics. It is a read-write transaction and includes non-primary key access to the customer table.
- **Order-Status** The order-status transaction queries the status of a customer's last order. This transaction also includes a query that uses non-primary key access to the customer table.
- **Delivery** The delivery transaction consists of processing a batch of 10 new orders. Each order is processed in full within the scope of a read-write database transaction.
- **Stock-Level** The stock-level transaction determines the number of recently sold items that have a stock-level below a specified threshold. It is categorized as a heavy read-only database transaction as it involves tables join.

5.2.2 Distributed TPC-C

We have a distributed version in which clients/terminals are able to execute in different machines as presented in [Pre+08] . It uses a "master console" that from a java property file, reads the JDBC driver url that is to be used and the desired workload. After reading the file, the "master console" creates an UDP server that waits for terminals to connect. The master receives as parameter the number of terminals that should wait for before sending the start signal. This signal exists in order to make all terminals start the benchmark at the same time.

TPC-C terminals receive by parameter the IP of the "master console" and communicate with the master through that connection. Upon connection the master sends all the info that read from the file to the terminal, so that the terminal is able to initiate its database connection. The workload is sent along with the connection info in order for the terminal to determine which operations to execute and how frequently. In the end

of the benchmark, each terminal sends its results back to the master, which in turn processes those results and displays the statistics of the benchmark having the results of all distributed terminals in consideration.

5.3 Experimental Setup

In the next sections, we will present some results that shows how our solution performs.

The experiments were run in a cluster with 8 nodes that are connected through a private Gigabit ethernet. Hardware wise, the nodes are heterogeneous, i.e, present some differences. Nodes 1 to 5 are equipped with 2xQuad-core AMD Opteron 2376, each core running at 2.3Ghz and with 4x512KB L2 cache size. These same nodes have a total amount of 16GB of physical memory available. Hereafter, these nodes are known as Group 1.

Nodes 6 to 8 are equipped with 1xQuad-core Intel(R) Xeon(R) CPU X3450, each running at 2.66Ghz and with hyperthreading enabled. These nodes have a total amount of 8GB of physical memory available. Hereafter, these nodes are known as Group 2. In terms of software all nodes run Linux 2.6.26-2-amd64, Debian 5.0.10. The Java version used was version 6, OpenJDK (Iced-Tea6 1.8.10), package 6b18-1.8.10-0 lenny2. For group communication we used JGroups 3.3.0.Final version.

Unless stated otherwise, all tests were executed five times removing the best and worst result. The final result is the average of the remaining values. Throughput results are presented in Transactions per Minute (TPM).

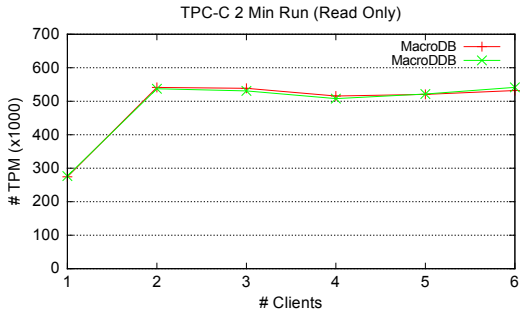
5.4 Overhead

The proposed solution works as a *middleware* that allows to replicate Macro-components. As such, it is natural for some overhead to exist, compared to the standalone case. The replication component does some extra work, namely, when write transactions exist, such as conflict checking, network communication and generating write sets.

To evaluate the overhead we compare MacroDB with MacroDDB with a single network node (both using H2 replicas). Both systems will be tested under different TPC-C workloads and have three replicas (one primary and two secondaries) in the underlying Macro-component configuration. A machine from Group 1 was used for these tests.

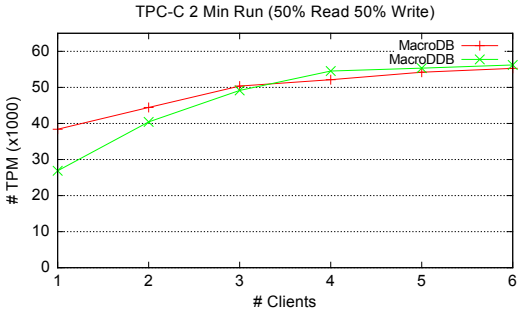
In Figure 5.2(a) we measure the overhead when there is only read-only transactions. The results demonstrate that our solution nearly has no overhead with this type of workload. This is because, as stated before, with read-only transactions there is no need for network communication nor write set handling, with MacroDDB working mainly as a proxy to the underlying Macro-component.

In Figure 5.2(b), with 20% write transaction, we already noticed some overhead. Although this workload has a reduced percentage of writes, it already shows how communication and write set handling, which is done by analyzing SQL statements, affects



(a) Read-only overhead results

(b) 80% read 20% write overhead results



(c) 50% read 50% write overhead results

Figure 5.2: Overhead measurement with different TPC-C workloads using H2 in-memory database

throughput. One observation is the fact that the MacroDDB system manages to have a better throughput when the number of clients is increased. This is due to the fact that our system, besides conflicts, also checks for deadlocks. The standalone MacroDB has no such checks and neither the underlying in-memory database in *MVCC* mode. The database engine deals with deadlocks resorting to timeouts. By waiting for the timeout (with the value of 1 second) the database won't process any other request, thus reducing the throughput. Focusing on the one client configuration, the overhead was of 20% compared to standalone MacroDB. This was the highest overhead observed on this benchmark.

Finally, in Figure 5.2(c) with 50% write transactions, the communication system and statement parser were more heavily used. We expected for the associated overheads to increase, and that was what happened. Focusing again in the one client configuration, which presented the highest overhead for the benchmark, shows an overhead of 30%.

To demonstrate how group communication and SQL statement parsing affects performance, we ran a test where only a single client and a single network node existed under the 80-20 TPC-C workload. Three software configurations were used: the first with the communication system and the statement parsing enabled, i.e, with the current MacroDDB operation model; in the second we removed the update propagation, this is not necessary with a single node; and third a configuration without communication and

SQL statement parsing, which still leads to a correct execution when we have a single client and a single network node, as no communication nor write set handling is necessary.

Figure 5.3: Task overhead

Figure 5.3 demonstrates that both SQL statement parsing and group communication impose most part of the overhead, supporting our previous claim. Unfortunately, network communication cost is a price that the system must pay for replication, as network nodes need to communicate to propagate updates among them. The only way to reduce this source of overhead is by minimizing the number of times that a given application resorts to network communication and minimizing the size of sent messages. On the other hand SQL statement parsing also presents a significant overhead. In this case, there are other techniques to retrieve the transaction's write set, such as the use of triggers. In our solution we did not explore that path, but may be a point worth considering in the future. Overall, the gains in performance by removing both communication and statement parsing is similar to the overhead observed in the results of overhead for this workload (Figure 5.2(b)), i.e, 20%.

5.5 Combining Multi-Master with Primary-Secondary Approach

In our approach we combine a multi-master approach among network nodes, with a primary-secondary replication approach in each machine. In this section we compare this hierarchical replication scheme with a flat multi-master replication scheme, in which all replicas are master (i.e., can process write transactions). To compare both replication approaches, we ran in four nodes, two configurations: *MacroDDB*, with one MacroDDB with three replicas running in each machine; *MacroDDB-Flat*, with two MacroDDB with one replica each in each machine. *MacroDDB-Flat* simulates a simple multi-master replication solution, as a single replica exists in each MacroDB. To this end, we ran the experiments with group 1 machines, unless stated otherwise.

5.5.1 Memory

We start by analyzing the memory consumption of both approaches. Figure 5.4 illustrates that result.

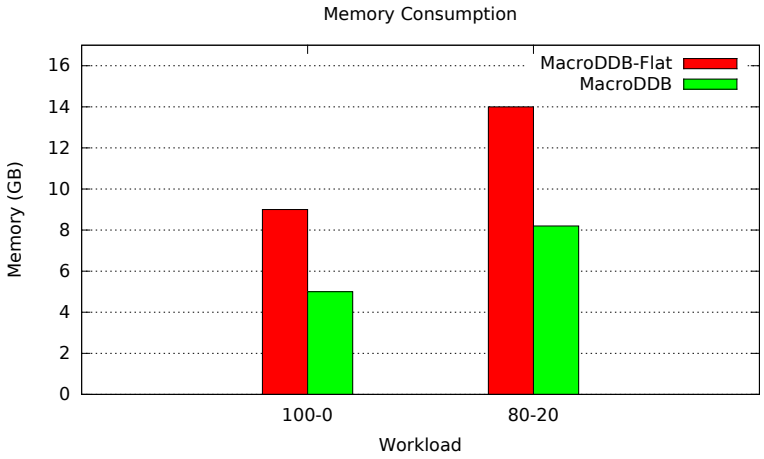
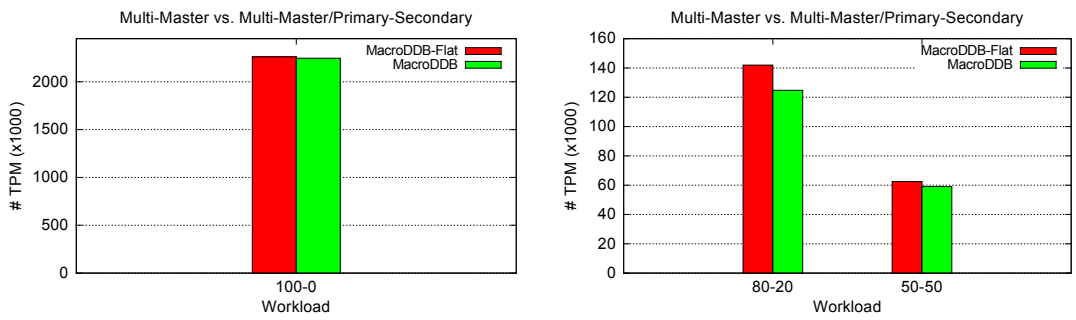


Figure 5.4: Memory usage

With our approach, the MacroDB component is able to share some objects among database replicas because they are executing in the same *JVM*. With this, it is obvious that our solution is much more memory efficient than the MacroDDB-Flat approach. In the MacroDDB-Flat approach, as each master runs in a separate *JVM*, they are unable to share objects among replicas. This significantly increases memory consumption in a given physical machine. Furthermore, we were unable to increase the dataset size in the MacroDDB-Flat approach without starting swaping during the benchmark. This is one point that favors the MacroDDB approach as there is some margin even in our test machines to increase the dataset size.

5.5.2 Throughput

In this section we present throughput results.



(a) Read-only workload

(b) Write workloads

Figure 5.5: MacroDDB vs. MacroDDB-Flat

In the read-only benchmark presented in Figure 5.5(a) both setups behaved similarly as there is no communication. There is only the downside of the MacroDDB-Flat approach consuming more memory than the MacroDDB approach.

In Figure 5.5(b) we present workloads with write operations. In this case, the MacroDDB-Flat approach has a better throughput. MacroDDB hierarchical replication was expected to generate less communication overhead (by forming smaller groups), and less computation (as less nodes need to execute transaction validation), one would expect it to perform better than the MacroDDB-Flat approach. To understand these results, we further analyzed these experiments and we found out the explanation for the results.

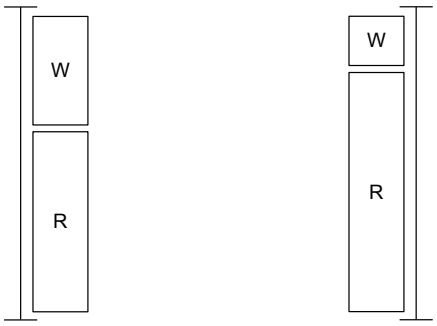


Figure 5.6: Write operation abort consequences

With the MacroDDB-Flat approach one of the heaviest write transaction aborted frequently: 70% aborted transactions in the 80-20 workload and 76% aborted transactions in the 50-50 workload. These values compare with 10% aborted transactions in the 80-20 workload and 15% aborted transactions in the 50-50 workload when using MacroDDB. Aborting a write transaction earlier reduces the load in the machine and, by avoiding expensive communication allows more time for executing quick read transactions. This allows to increase the overall number of committed transactions but with a different ratio of read and write transactions. Such situation is depiicted in Figure 5.6.

Table 5.1: Ratio between read-only/read-write transactions

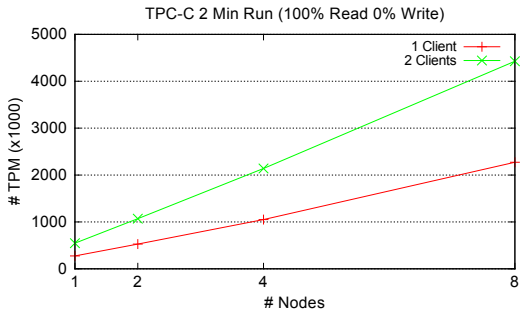
Workload	System	Read/Write ratio
80-20	MacroDDB	4.82
	MacroDDB-Flat	5.64
50-50	MacroDDB	1.30
	MacroDDB-Flat	1.50

On the 50-50 workload the gap is smaller because the chance to execute a cheaper transaction after an abort is smaller (as in half of the cases a new write transaction is scheduled), thus the gains are smaller. Table 5.1 summarizes the differences between the MacroDDB approach and the MacroDDB-Flat approach by presenting ratio between read-only and read-write transactions. The ratios are consistent with the presented graph and it shows that the MacroDDB-Flat approach does more read-only transactions than

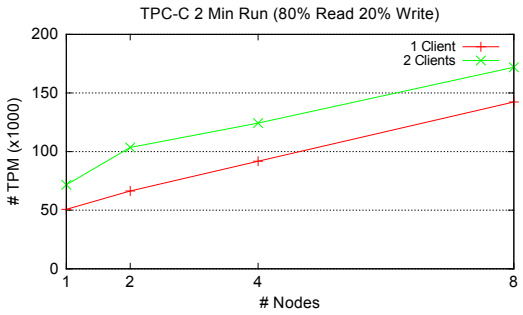
the MacroDDB approach and that it presents a larger deviation from the used workload.

5.6 Scalability

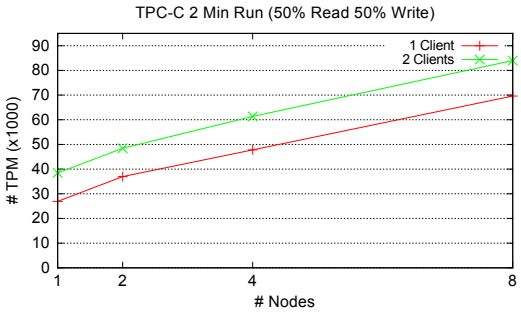
This section evaluates the scalability of MacroDDB. We started by running a configuration where each network node has a single client associated as means to measure how MacroDDB scales with the increase of the number of machines, and afterwards we doubled the number of clients at each configuration to evaluate how does the MacroDDB scales with the increase of clients. The underlying MacroDB used a configuration with three replicas (one primary and two secondaries). The tests used all available nodes from both Group 1 and 2. The results are shown in Figure 5.7.



(a) Read-only scalability results



(b) 80% read 20% write scalability results



(c) 50% read 50% scalability results

Figure 5.7: Scalability results using TPC-C and different workloads

We start by analyzing the read-only case which is shown in Figure 5.7(a). With read-only transactions there is no communication nor statement parsing overhead and the system is basically directly using the underlying MacroDB. As such we are taking full advantage of the MacroDB possibilities and exploring the scalability that running in several nodes provides, therefore increasing overall system throughput. At each network node configuration, the throughput in the two clients per node is doubled compared with the one client case.

The results for the 80-20 workload presented in Figure 5.7(b) shows that the gains

in throughput when moving from the one client configuration to the two client configuration are not as high as in the read-only case, mainly because read-write transactions start interfering with each other and may need to wait in order to proceed. Even when this is the case the system managed to increase its throughput in about 40% in average compared to the one client case. In the two network nodes configuration, the system managed an increase of 50%. We can still see that the system also manages to scale with the increase of network nodes.

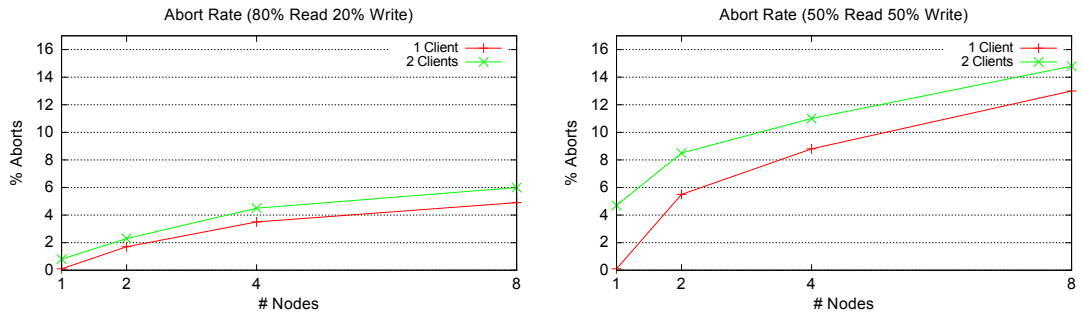
Finally, Figure 5.7(c) presents the results for 50-50 workload. In average and with this workload we managed to improve the throughput by 30%. The maximum improvement observed was of 40% in the first configuration. In this case, the system managed as well to scale as the number of network nodes increased, which is consistent with the remaining workloads. To finalize these benchmarks, Table 5.2 summarizes the results obtained in this section.

Table 5.2: Scalability summary

Workload	Average Improvement
100-0 (or read-only)	100%
80-20	40%
50-50	30%

5.6.1 Abort Rate

One final metric to be evaluated is the abort rate. Basically, with this metric we try to evaluate what is the impact of the number of nodes and client in the success of transactions. The results are shown in Figure 5.8. It should be noted that the benchmark already forces an abort of 1% in one of the transactions (when write operation exists). That abort rate is counted for on the presented results. Read-only transactions has been omitted because there are no aborted transactions as expected.



(a) 80% read 20% write workload (b) 50% read 50% write workload

Figure 5.8: Transactions Abort Rate

Starting with 20% write load presented in Figure 5.8(a), we start with a 0.1% abort rate

in the one client per node configuration. The aborts in this case are the ones predicted by the benchmark. As a reminder, the presented abort rate counts for conflicts following the snapshot isolation rules and the deadlocks that we detected (to avoid timeout by the database). The deadlock detection is not perfect and aborts the first transaction that runs into the deadlock, which means that in the worst case both conflicting transactions get aborted. When the number of network nodes is increased, the two client configuration keeps a steady increase compared to the one client configuration. With the 80-20 workload the abort rate reaches a maximum value of 5.8% when sixteen clients are running in the system, i.e, eight nodes with two clients each.

Moving on into the 50% write load benchmark presented in Figure 5.8(b) we noticed an increase in the abort rate which was expected. With this kind of write rate, the transactions interactions become more frequent and therefore deadlock and snapshot isolation conflicts become more likely. The lowest abort rate observed was in the one node with one client configuration having 0.1% of abort rate, while the highest was of 14.8% in the eight nodes with two clients each configuration.

5.6.2 Taking Advantage of Multi-cores

Ultimately, one of the goals is to take advantage of multi-core architectures in the distributed setting.

The read-only case is omitted because it trivially scales as demonstrated in Figure 5.7(a). Figure 5.9 compares MacroDB (using 3 replicas) with MacroDB with only one replica on the 80-20 workload. The figure is presented to show that although the benchmark has a low percentage of writes, it needs more than two clients to really surpass MacroDB with one replica. This is expected as with a single client, there is no concurrent operations being performed and MacroDB ends up leading to more work, as write transactions need to execute in all replicas.

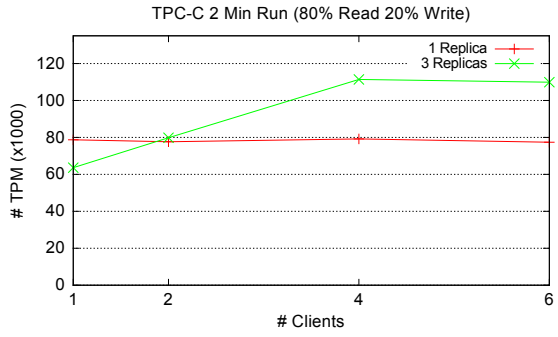
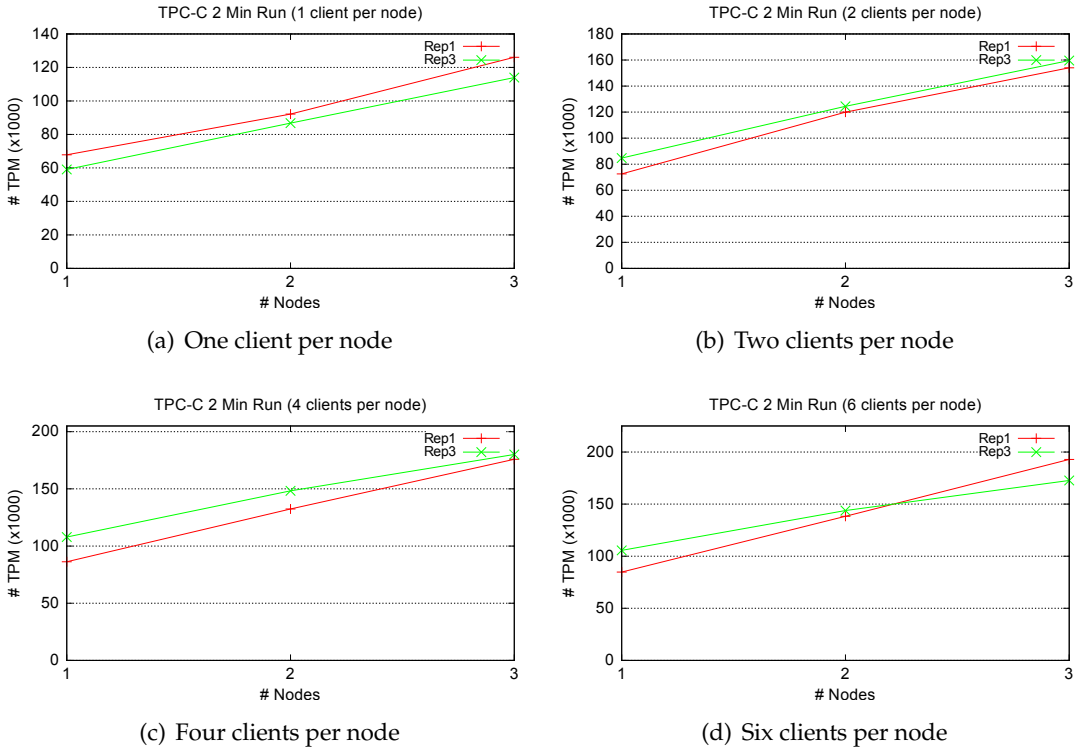


Figure 5.9: MacroDB performance sample

To determine if the system takes advantage of multi-core systems, we ran the TPC-C benchmark in group 2 nodes and with two setups: first setup is composed by a MacroDDB instance with one replica running in the node simulating the standalone database; the second setup is composed by a MacroDDB instance with three replicas. We vary the

number of participating nodes and the number of clients. The results are illustrated in Figure 5.10. Each graph has a different number of clients per node.



(a) One client per node

(b) Two clients per node

(c) Four clients per node

(d) Six clients per node

Figure 5.10: Multi-core results

Figure 5.10(a) presents the results when there is one client per node. The Rep1 configuration managed to have better throughput than the MacroDB setup, but it is consistent with the centralized context as shown in Figure 5.9. At the end, the gap is wider because the standalone database setup starts suffering from the situation depicted in Figure 5.6, which is the increase of the abort rate on write transactions. Having a small number of nodes helps keeping the comparison as fair as possible by reducing transaction conflicts, although when the number of clients increase it is likely for more aborts to occur.

Increasing the number of clients in Figure 5.10(b), we observe that the Rep3 configuration starts by having better throughput, which shows that the designed system is able to take advantage of multi-core architectures in the distributed setting even when communication overhead limits cpu usage.

In Figure 5.10(c) the MacroDDB system managed to obtain the highest increase, mainly because the cpu spends less time idle. At the three nodes mark we verify the same effect of the abort rate increasing with the Rep1 configuration. This is consistent in all presented results of Figure 5.10. Even then, the Rep3 configuration was able to have a slightly better throughput while having a higher rate of commits.

When we move to the configuration where each node has six clients, the underlying MacroDB seems to have reached its peak performance, as there was no notable increase

in throughput. On the other hand the increase in clients quickly augmented the abort rate in the Rep1 configuration, thus the faster throughput increase.

Overall the MacroDDB system managed to take advantage of multi-core architectures in the distributed setting. With more clients the system should scale, as long as there are enough secondary replicas and cpu cores to attend to all client's (read) requests. Table 5.3 summarizes the results obtained in this section. The average improvement counts for all data points in the graph, while the best improvement measures the best throughput improvement.

Table 5.3: Multi-core usage summary

# Clients (per node)	Average Improvement (%)	Best Improvement (%)
1	-8%	-5%
2	6%	16%
4	11%	24%
6	3%	22%

6

Conclusion

6.1 Concluding Remarks

Replication and distribution is a powerful technique to improve availability by masking faults and making data available in a set of different nodes in the network and/or to increase performance by distributing load across nodes. Having multi-core architectures in these nodes can help improve performance, but this topic still needs further research and practical solutions.

The Macro-component [Mar10] is an abstraction that was developed to take advantage of multi-core systems in the centralized context by taking advantage of standalone implementations and combining them to distribute work or even executing jobs in parallel among other possible tasks. This is achieved in a way that is transparent to the application.

This dissertation proposes an extension to the Macro-component that enables the use of Macro-components in a distributed setting efficiently. To this end, we propose the design of a replication *middleware* for Macro-components. The solution is generic and allows to replicate any Macro-component, while providing the same interface for applications using this new distributed Macro-component.

Using the generic system, we have built a prototype that focused on replicating in-memory databases. In this case, applications continue to use the Java JDBC interface to access the database, oblivious of the distributed and replicated nature of the underlying database.

The MacroDDB uses MacroDB as its underlying Macro-component and uses a hierarchical replication algorithm combining multi-master replication across node and primary-backup in each node. This way instead of having all Macro-component replicas participating in the communication protocol, we have the primary replica responsible for that job. Updates are asynchronously propagated to the secondary replicas.

We evaluated our system using a standard benchmark, namely the TPC-C benchmark. Our evaluation shows that our solution is able to scale as the number of clients and nodes increase and takes advantage of multi-core machines. It also shows that the presented solution is resource efficient when multiple replicas are deployed in a single machine.

In conclusion the presented solution allows for an easy and efficient way to replicate Macro-components, thus allowing to take advantage of multi-core systems in the distributed setting.

6.2 Future Work

During the course of this work, several different directions for future work have been uncover. Among these, the most interesting are the following:

- Using partial replication as an extension of this work, namely in these in-memory structures where a given node may be scarce in memory resources, partial replication would help dealing with this problem. The implemented solution may be used to perform this study;
- To improve replication protocol, namely by requiring less expensive communication primitives, one might use the Conflict Free Replicated Datatypes (CRDTs) [Sha+11] which never conflict and always merge into a consistent state;
- Explore the use of triggers to extract the transaction's write set and compare with the SQL statement parsing method.
- The use of other real world benchmarks to further understand how well does this solution fares.

Bibliography

[Agr+97] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. "Exploiting Atomic Broadcast in Replicated Databases (Extended Abstract)". In: *Proceedings of the Third International Euro-Par Conference on Parallel Processing*. Euro-Par '97. London, UK, UK: Springer-Verlag, 1997, pages 496–503. ISBN: 3-540-63440-1. URL: http://dl.acm.org/citation.cfm?id=646662.699395.

[Ama08] Amazon. *Amazon S3 Availability Event: July 20, 2008*. 2008. URL: http://status.aws.amazon.com/s3-20080720.html.

[Ami+00] Y. Amir, C. Danilov, and S. Stanton. "A low latency, loss tolerant architecture and protocol for wide area group communication". In: *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*. 2000, pages 327–336. DOI: 10.1109/ICDSN.2000.857557.

[AIn+08] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. G. de Mendívil, and F. D. Muñoz Escoí. "SIPRe: a partial database replication protocol with SI replicas". In: *Proceedings of the 2008 ACM symposium on Applied computing*. SAC '08. Fortaleza, Ceara, Brazil: ACM, 2008, pages 2181–2185. ISBN: 978-1-59593-753-7. DOI: 10.1145/1363686.1364207. URL: http://doi.acm.org/10.1145/1363686.1364207.

[Asa+09] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. "A view of the parallel computing landscape". In: *Commun. ACM* 52.10 (Oct. 2009), pages 56–67. ISSN: 0001-0782. DOI: 10.1145/1562764.1562783. URL: http://doi.acm.org/10.1145/1562764.1562783.

[Ban98] B. Ban. "Design and implementation of a reliable group communication toolkit for java". In: *Cornell University* (1998).

[Bra+85] G. Bracha and S. Toueg. "Asynchronous consensus and broadcast protocols". In: *J. ACM* 32.4 (Oct. 1985), pages 824–840. ISSN: 0004-5411. DOI: 10.

1145/4221.214134. URL: http://doi.acm.org/10.1145/4221.214134.

[Bud+93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. "Distributed systems (2nd Ed.)" In: edited by S. Mullender. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993. Chapter The primary-backup approach, pages 199–216. ISBN: 0-201-62427-3. URL: http://dl.acm.org/citation.cfm?id=302430.302438.

[Cac+06] J. a. Cachopo and A. Rito-Silva. "Versioned boxes as the basis for memory transactions". In: *Sci. Comput. Program.* 63.2 (Dec. 2006), pages 172–185. ISSN: 0167-6423. DOI: 10.1016/j.scico.2006.05.009. URL: http://dx.doi.org/10.1016/j.scico.2006.05.009.

[Cam+07] L. Camargos, F. Pedone, and M. Wieloch. "Sprint: a middleware for high-performance transaction processing". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pages 385–398. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273036. URL: http://doi.acm.org/10.1145/1272996.1273036.

[Cas+99] M. Castro and B. Liskov. "Practical Byzantine fault tolerance". In: *Proceedings of the third symposium on Operating systems design and implementation*. OSDI '99. New Orleans, Louisiana, United States: USENIX Association, 1999, pages 173–186. ISBN: 1-880446-39-1. URL: http://dl.acm.org/citation.cfm?id=296806.296824.

[Cas+00] M. Castro and B. Liskov. "Proactive recovery in a Byzantine-fault-tolerant system". In: *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*. OSDI'00. San Diego, California: USENIX Association, 2000, pages 19–19. URL: http://dl.acm.org/citation.cfm?id=1251229.1251248.

[Cas+03] M. Castro, R. Rodrigues, and B. Liskov. "BASE: Using abstraction to improve fault tolerance". In: *ACM Trans. Comput. Syst.* 21.3 (Aug. 2003), pages 236–269. ISSN: 0734-2071. DOI: 10.1145/859716.859718. URL: http://doi.acm.org/10.1145/859716.859718.

[Cor+12] J. C. Corbett *et al.* "Spanner: Google's globally-distributed database". In: *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pages 251–264. ISBN: 978-1-931971-96-6. URL: http://dl.acm.org/citation.cfm?id=2387880.2387905.

[Cou+09] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. "D2STM: Dependable Distributed Software Transactional Memory". In: *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. PRDC

'09. Washington, DC, USA: IEEE Computer Society, 2009, pages 307–313. ISBN: 978-0-7695-3849-5. DOI: 10.1109/PRDC.2009.55. URL: http://dx.doi.org/10.1109/PRDC.2009.55.

[Cou+11] M. Couceiro, P. Romano, and L. Rodrigues. "PolyCert: polymorphic self-optimizing replication for in-memory transactional grids". In: *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*. Middleware'11. Lisbon, Portugal: Springer-Verlag, 2011, pages 309–328. ISBN: 978-3-642-25820-6. DOI: 10.1007/978-3-642-25821-3_16. URL: http://dx.doi.org/10.1007/978-3-642-25821-3_16.

[Cow+06] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. "HQ replication: a hybrid quorum protocol for byzantine fault tolerance". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. OSDI '06. Seattle, Washington: USENIX Association, 2006, pages 177–190. ISBN: 1-931971-47-1. URL: http://dl.acm.org/citation.cfm?id=1298455.1298473.

[DeC+07] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: amazon's highly available key-value store". In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pages 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[Déf+04] X. Défago, A. Schiper, and P. Urbán. "Total order broadcast and multicast algorithms: Taxonomy and survey". In: *ACM Comput. Surv.* 36.4 (Dec. 2004), pages 372–421. ISSN: 0360-0300. DOI: 10.1145/1041680.1041682. URL: http://doi.acm.org/10.1145/1041680.1041682.

[Dis+11] T. Distler and R. Kapitza. "Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency". In: *Proceedings of the sixth conference on Computer systems*. EuroSys '11. Salzburg, Austria: ACM, 2011, pages 91–106. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966455. URL: http://doi.acm.org/10.1145/1966445.1966455.

[Eln+05] S. Elnikety, W. Zwaenepoel, and F. Pedone. "Database Replication Using Generalized Snapshot Isolation". In: *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*. SRDS '05. Washington, DC, USA: IEEE Computer Society, 2005, pages 73–84. ISBN: 0-7695-2463-X. DOI: 10.1109/RELDIS.2005.14. URL: http://dx.doi.org/10.1109/RELDIS.2005.14.

[Eln+06] S. Elnikety, S. Dropsho, and F. Pedone. "Tashkent: uniting durability with transaction ordering for high-performance scalable database replication".

In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. EuroSys '06. Leuven, Belgium: ACM, 2006, pages 117–130. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217947. URL: http://doi.acm.org/10.1145/1217935.1217947.

[Gar+93] J. A. Garay and Y. Moses. "Fully polynomial Byzantine agreement in t + 1 rounds". In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. STOC '93. San Diego, California, United States: ACM, 1993, pages 31–41. ISBN: 0-89791-591-7. DOI: 10.1145/167088.167101. URL: http://doi.acm.org/10.1145/167088.167101.

[Ghe+03] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google file system". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pages 29–43. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945450. URL: http://doi.acm.org/10.1145/945445.945450.

[Gra+96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. "The dangers of replication and a solution". In: *SIGMOD Rec.* 25.2 (June 1996), pages 173–182. ISSN: 0163-5808. DOI: 10.1145/235968.233330. URL: http://doi.acm.org/10.1145/235968.233330.

[Har+08] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. "OLTP through the looking glass, and what we found there". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pages 981–992. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376713. URL: http://doi.acm.org/10.1145/1376616.1376713.

[Har+05] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. "Composable memory transactions". In: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '05. Chicago, IL, USA: ACM, 2005, pages 48–60. ISBN: 1-59593-080-9. DOI: 10.1145/1065944.1065952. URL: http://doi.acm.org/10.1145/1065944.1065952.

[JP+02] R. Jiménez-Peris, M. Patiño Martínez, and G. Alonso. "Non-Intrusive, Parallel Recovery of Replicated Data". In: *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*. SRDS '02. Washington, DC, USA: IEEE Computer Society, 2002, pages 150–. ISBN: 0-7695-1659-9. URL: http://dl.acm.org/citation.cfm?id=829526.831118.

[Kal+08] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. "H-store: a high-performance, distributed main memory transaction processing

system". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pages 1496–1499. ISSN: 2150-8097. URL: http://dl.acm.org/citation.cfm?id=1454159.1454211.

[Kap+12] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. "All about Eve: execute-verify replication for multi-core servers". In: *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pages 237–250. ISBN: 978-1-931971-96-6. URL: http://dl.acm.org/citation.cfm?id=2387880.2387903.

[Kem+98] B. Kemme and G. Alonso. "A Suite of Database Replication Protocols based on Group Communication Primitives". In: *Proceedings of the The 18th International Conference on Distributed Computing Systems*. ICDCS '98. Washington, DC, USA: IEEE Computer Society, 1998, pages 156–. ISBN: 0-8186-8292-2. URL: http://dl.acm.org/citation.cfm?id=850926.851673.

[Kem+00] B. Kemme and G. Alonso. "A new approach to developing and implementing eager database replication protocols". In: *ACM Trans. Database Syst.* 25.3 (Sept. 2000), pages 333–379. ISSN: 0362-5915. DOI: 10.1145/363951.363955. URL: http://doi.acm.org/10.1145/363951.363955.

[Kot+07] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. "Zyzzyva: speculative byzantine fault tolerance". In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pages 45–58. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294267. URL: http://doi.acm.org/10.1145/1294261.1294267.

[Lam98] L. Lamport. "The part-time parliament". In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pages 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[Lam+84] L. Lamport and M. Fischer. *Byzantine Generals and transaction commit protocols*. Technical report Op. 62. SRI International and Yale University, 1984.

[Mal+97] D. Malkhi and M. Reiter. "Unreliable Intrusion Detection in Distributed Computations". In: *Proceedings of the 10th IEEE workshop on Computer Security Foundations*. CSFW '97. Washington, DC, USA: IEEE Computer Society, 1997, pages 116–. ISBN: 0-8186-7990-5. URL: http://dl.acm.org/citation.cfm?id=794197.795085.

[Man+13] R. Manning and L. Francalanci. *JSqlParser, SQL statement parser*. http://jsqlparser.sourceforge.net/. May 2013.

[Mar10] P. Mariano. "RepComp - Replicated Software Components for Improved Performance". Master's thesis. Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2010.

[Pel+12] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. "When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication". In: *Proceedings of the 2012 IEEE 32nd International Conference on Distributed Computing Systems*. ICDCS '12. Washington, DC, USA: IEEE Computer Society, 2012, pages 455–465. ISBN: 978-0-7695-4685-8. DOI: 10.1109/ICDCS.2012.55. URL: http://dx.doi.org/10.1109/ICDCS.2012.55.

[Pin01] A. Pinto. "Appia: A Flexible Protocol Kernel Supporting Multiple Coordinated Channels". In: *Proceedings of the The 21st International Conference on Distributed Computing Systems*. ICDCS '01. Washington, DC, USA: IEEE Computer Society, 2001, pages 707–. URL: http://dl.acm.org/citation.cfm?id=876878.879271.

[Pla+04] C. Plattner and G. Alonso. "Ganymed: scalable replication for transactional web applications". In: *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Middleware '04. Toronto, Canada: Springer-Verlag New York, Inc., 2004, pages 155–174. ISBN: 3-540-23428-4. URL: http://dl.acm.org/citation.cfm?id=1045658.1045671.

[Pre+08] N. M. Preguiça, R. Rodrigues, C. Honorato, and J. Lourenço. "Byzantium: Byzantine-Fault-Tolerant Database Replication Providing Snapshot Isolation". In: *HotDep*. 2008.

[Sal+06] J. Salas, F. Perez-Sorrosal, M. Patiño Martínez, and R. Jiménez-Peris. "WS-replication: a framework for highly available web services". In: *Proceedings of the 15th international conference on World Wide Web*. WWW '06. Edinburgh, Scotland: ACM, 2006, pages 357–366. ISBN: 1-59593-323-9. DOI: 10.1145/1135777.1135831. URL: http://doi.acm.org/10.1145/1135777.1135831.

[Sal+11] T.-I. Salomie, I. E. Subasu, J. Giceva, and G. Alonso. "Database engines on multicores, why parallelize when you can distribute?" In: *Proceedings of the sixth conference on Computer systems*. EuroSys '11. Salzburg, Austria: ACM, 2011, pages 17–30. ISBN: 978-1-4503-0634-8. DOI: 10.1145/1966445.1966448. URL: http://doi.acm.org/10.1145/1966445.1966448.

[Sch+10] N. Schiper, P. Sutra, and F. Pedone. "P-Store: Genuine Partial Replication in Wide Area Networks". In: *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*. SRDS '10. Washington, DC, USA: IEEE Computer Society, 2010, pages 214–224. ISBN: 978-0-7695-4250-8. DOI: 10.1109/SRDS.2010.32. URL: http://dx.doi.org/10.1109/SRDS.2010.32.

[Sch93] F. B. Schneider. "Distributed systems (2nd Ed.)" In: edited by S. Mullender. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993. Chapter Replication management using the state-machine approach, pages 169–

197. ISBN: 0-201-62427-3. URL: http://dl.acm.org/citation.cfm?id=302430.302437.

[Ser+07] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme. "Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation". In: *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*. PRDC '07. Washington, DC, USA: IEEE Computer Society, 2007, pages 290–297. ISBN: 0-7695-3054-0. DOI: 10.1109/PRDC.2007.39. URL: http://dx.doi.org/10.1109/PRDC.2007.39.

[Sha+11] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-free replicated data types". In: *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pages 386–400. ISBN: 978-3-642-24549-7. URL: http://dl.acm.org/citation.cfm?id=2050613.2050642.

[Sha+95] N. Shavit and D. Touitou. "Software transactional memory". In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. PODC '95. Ottowa, Ontario, Canada: ACM, 1995, pages 204–213. ISBN: 0-89791-710-3. DOI: 10.1145/224964.224987. URL: http://doi.acm.org/10.1145/224964.224987.

[Sin+09] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. "Zeno: eventually consistent Byzantine-fault tolerance". In: *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. NSDI'09. Boston, Massachusetts: USENIX Association, 2009, pages 169–184. URL: http://dl.acm.org/citation.cfm?id=1558977.1558989.

[Soa+13] J. Soares, J. Lourenço, and N. M. Preguiça. "MacroDB: Scaling Database Engines on Multicores". In: *Proceedings of Euro-Par 2013*. 2013.

[Tra13] Transaction Processing Council. *TPC-C, On-line transaction processing benchmark*. 2013. URL: http://www.tpc.org/tpcc/default.asp.

[Val12] T. Vale. "A Modular Distributed Transactional Memory Framework". Master's thesis. Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2012.

[Vog09] W. Vogels. "Eventually consistent". In: *Commun. ACM* 52.1 (Jan. 2009), pages 40–44. ISSN: 0001-0782. DOI: 10.1145/1435417.1435432. URL: http://doi.acm.org/10.1145/1435417.1435432.

[Wes+09] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. "Tolerating latency in replicated state machines through client speculation". In: *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. NSDI'09. Boston, Massachusetts: USENIX Association, 2009, pages 245–260. URL: http://dl.acm.org/citation.cfm?id=1558977.1558994.

[Wie+00a] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. "Understanding Replication in Databases and Distributed Systems". In: *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*. ICDCS '00. Washington, DC, USA: IEEE Computer Society, 2000, pages 464–. ISBN: 0-7695-0601-1. URL: http://dl.acm.org/citation.cfm?id=850927.851782.

[Wie+00b] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso. "Database Replication Techniques: A Three Parameter Classification". In: *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems*. SRDS '00. Washington, DC, USA: IEEE Computer Society, 2000, pages 206–. ISBN: 0-7695-0543-0. URL: http://dl.acm.org/citation.cfm?id=829525.831077.

[Wie+05] M. Wiesmann and A. Schiper. "Comparison of Database Replication Techniques Based on Total Order Broadcast". In: *IEEE Trans. on Knowl. and Data Eng.* 17.4 (Apr. 2005), pages 551–566. ISSN: 1041-4347. DOI: 10.1109/TKDE.2005.54. URL: http://dx.doi.org/10.1109/TKDE.2005.54.