Luís Miguel Dias Rocha

Bachelor in Computer Science and Engineering

Ginger: A Transactional Middleware with Data and Operation Centric Mixed Consistency

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Hervé Miguel Cordeiro Paulino,
Associate Professor, Computer Science Department,
NOVA School of Science and Technology, NOVA
University of Lisbon

Examination Committee:

Chair: João Baptista da Silva Araújo Júnior, Associate Professor, NOVA School of Science and Technology, NOVA University of Lisbon

Members: Francisco Cipriano da Cunha Martins, Associate Professor, Faculty of Sciences and Technology, University of the Azores

Hervé Miguel Cordeiro Paulino, Associate Professor, NOVA School of Science and Technology, NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

November, 2020

Ginger: A Transactional Middleware with Data and Operation Centric Mixed Consistency

Copyright © Luís Miguel Dias Rocha, NOVA School of Science and Technology, NOVA University Lisbon.
The NOVA School of Science and Technology and the NOVA University Lisbon have the right, perpetual and without geographical boundaries, to file and publish this dissertation through printed copies reproduced on paper or on digital form, or by any other means known or that may be invented, and to disseminate through scientific repositories and admit its copying and distribution for non-commercial, educational or research purposes, as long as credit is given to the author and editor.

To my parents, my sisters and my friends.

Acknowledgements

Firstly I want to greatly thank my advisor, professor Hervé Paulino, for the restless and constant support throughout the duration of the development of this dissertation, and for, in a year which has been stressful and full of work for everybody due to the current world circumstances, finding time to always help with the problems that arose. I also thank João Silva, for always being there to help me and provide his knowledge on the area of expertise of this dissertation. I'm very thankful to FCT-UNL and the Department of Informatics for being a second home to me and helping me grow and educate myself, not only as a student but as a human being.

This thesis was developed in the context of the project DeDuCe(PTDC/ CCI-COM/ 32166/ 2017) funded by Fundação para a Ciência e Tecnologia.

Everything I manage to achieve in life and the immense happiness I always possessed is greatly thanks to my family who always gave me so much love and affection and always cared deeply about my interests and letting me be whoever I wanted to be. I am deeply thankful to my parents, Luís and Cristina, for always being there for me, and supporting me in this journey, which is as much mine as it is theirs. To my sisters, a big thank you for always embarking on my crazy jokes and helping me bring fun to the family.

I also want to thank in a major way to my friend, Maria, who knows me better than I do and who when I was in my darkest moments, when i doubted myself and my knowledge, was always there to support me and made realize that there was more in me, that I could easily make it through and that she was there to fight alongside me.

To my friend João, thank you for always having my back through the tough moments and always being there to celebrate the best moments. To my friends Gabriel and Francisco, thank you for always being there to help me, and always finding time throughout this 4 years together to have fun and joke around. Also a thank you to all the friends I made throughout this 5 years for making this journey one of the best of my life.

Abstract

Many modern digital services to correspond to user demand need to offer high availability and low response times. To that end, a lot of digital services resort to geo-replicated distributed systems. These systems are deployed closer to users, splitting latency across multiple servers and allowing for faster access and communication. However, to accommodate these systems the data stores are also split up across multiple locations. Committing an operation is such systems requires coordination among the multiple replicas. These systems must allow data to be stored as fast as possible without breaking safety constraints of the developers systems.

There are three main approaches to define the level of consistency to be guaranteed when accessing the data: over data, over operations or over transactions. The problem with approaches such as consistency over data or consistency over transactions is that they are very limited, as they can result in operations that could be executed in lower consistency levels to be executed at higher consistency levels. Our approach to this problem is the conciliation of executing transactions while expressing consistency in both data and operations. We instantiate this proposition in a middleware system, called Ginger, that is deployed between the user and the data stores. Ginger benefits from all the other approaches, allowing for execution of transactions, that include operations with different levels of consistency, over data with different levels of consistency. This provides the benefits of the isolation from transactions while also providing the performance and control, that consistency defined over operations and consistency defined over data provide.

Our experimental results show that Ginger comparing to previously mentioned approaches, such as consistency over data and consistency over transaction, provides faster transaction committing speeds. Ginger serves as proof of concept that using consistency defined both over data and operations while using transactions is possible and may be a viable approach. Further development of the system will provide more functionalities, further evaluation, and a more in-depth comparison to other systems.

Keywords: Distributed Systems, Consistency, Middleware, Replication, Data Store

Resumo

Os serviços digitais modernos para corresponder às necessidades dos utilizadores precisam de oferecer alta disponibilidade e baixos tempos de resposta. Para tal, os serviços digitais recorrem a sistemas geo-replicados. Esses sistemas são implantados perto dos utilizadores, dividindo a latência entre servidores. No entanto, para acomodar esses sistemas, os serviços de armazenamentos de dados são divididos. O *commiting* de uma operação nesses sistemas requer coordenação entre múltiplas réplicas. Esses sistemas devem permitir que os dados sejam armazenados rapidamente, sem quebrar restrições de segurança.

Existem três abordagens principais para definir o nível de consistência a ser garantido durante o acesso aos dados: sobre dados, sobre operações ou sobre transacções. O problema com abordagens como consistência sobre dados ou sobre transacções é que são limitadas, podendo resultar em operações de níveis de consistência baixos serem executadas com níveis de consistência mais altos. A nossa abordagem a este problema é a conciliação da expressão de consistência tanto nos dados como nas operações. Instanciámos esta proposição num sistema de *middleware*, denominado Ginger, que é implantado entre o usuário e os serviços de armazenamentos de dados. O Ginger beneficia de todas as abordagens referidas, permitindo a execução de transacções, que incluem operações com diferentes níveis de consistência, sobre dados com diferentes níveis de consistência. Isto beneficia do isolamento das transacções, ao mesmo tempo que fornece o desempenho e o controle, que a consistência definida nas operações e a consistência definida nos dados fornecem.

Os nossos resultados experimentais mostram que o Ginger, em comparação com as outras abordagens, como por exemplo consistência sobre os dados e consistência sobre a transação, fornece velocidades de *commiting* de transacções mais rápidas. Ginger serve como prova de conceito de que o uso de transacções com níveis de consistência definidos sobre os dados e operações é possível e pode ser uma abordagem viável. O desenvolvimento futuro do sistema fornecerá mais funcionalidades, avaliação adicional e uma comparação mais aprofundada com outros sistemas.

Palavras-chave: Sistemas Distribuídos, Consistência, Middleware, Replicação

Contents

List of Figures xvii

List of Tables xix

List of Listings xxi

Glossary xxiii

Acronyms xxv

1 Introduction 1
- 1.1 Context and Motivation . . . 1
- 1.2 Problem . . . 2
- 1.3 Developed Solution . . . 3
- 1.4 Contributions . . . 4
- 1.5 Document Outline . . . 5

2 State of the art 7
- 2.1 High Availability . . . 7
  - 2.1.1 Requirements . . . 7
  - 2.1.2 Limitations . . . 8
  - 2.1.3 System Requirements . . . 8
- 2.2 Replication . . . 8
- 2.3 Consistency Levels . . . 9
  - 2.3.1 Strong Consistency . . . 9
  - 2.3.2 Eventual Consistency . . . 9
  - 2.3.3 Causal Consistency . . . 11
  - 2.3.4 Explicit Consistency . . . 12
  - 2.3.5 Hybrid Consistency . . . 12
- 2.4 Multiple Consistency Levels . . . 13
  - 2.4.1 Consistency Over and Granularity . . . 13
  - 2.4.2 Consistency Defining . . . 14
  - 2.4.3 Consistency Levels . . . 15
  - 2.4.4 Dependencies . . . 16

2.4.5 Implemented Over 16
2.4.6 Evaluation 16
2.5 Ensuring Causality 17
2.6 Controlling Eventual Consistency 19
2.6.1 Implementation 20
2.7 Ordering Events 20
2.7.1 Clocks 20
2.8 Conclusion 21

3 Ginger 23

3.1 Overview 23
3.2 System Architecture 24
3.3 Data Service 25
3.3.1 Service Definition 25
3.3.2 Service Implementation 27
3.3.3 Service Registration 27
3.4 Front-end API 29
3.4.1 Ginger Session 29
3.4.2 Transaction 30
3.4.3 Operation 30
3.4.4 Scope 33
3.4.5 Condition and Loop 35
3.4.6 Annotation Processor 36
3.5 Processing Transactions 40
3.5.1 Processing Operations 40
3.5.2 Processing Scopes 45
3.5.3 Processing Conditions 46
3.5.4 Processing Loops 47
3.6 Middleware 48
3.7 Back-end Communication 50

4 Evaluation 51

4.1 Objective 51
4.2 Methodology 52
4.3 Conducted tests 53
4.3.1 Functionality tests 53
4.3.2 Consistency tests 61

5 Conclusion 71

5.1 Conclusion 71
5.2 Future Work 72
5.2.1 System development 72

5.2.2 System improvements 73

References **75**

List of Figures

3.1 Architecture Diagram 24
3.2 Transaction graphical example of `Result` dependencies 42
3.3 Diagram representative of the processing of a transaction with dependencies resulting from usage of results from operations 43
3.4 Transaction graphical example of `ConsistencyLevelObject` dependencies 44
3.5 Diagram representative of the processing of a transaction with dependencies resulting from usage of `ConsistencyLevelObject` objects 44
3.6 Diagram representative of a Scope with three operations. 46
3.7 Diagram representative of the processing of a scope. 47
3.8 Diagram representative of the sequence of execution of a transaction. . . . 49

4.1 Results from higher transfer percentage test 67
4.2 Results from higher deposits/withdrawal percentage test 68
4.3 Results from higher balance check percentage test 69

List of Tables

2.1 Works addressing multiple consistency levels 13

3.1 API Annotations 38

4.1 Node specification 63

List of Listings

3.1 The DataService interface 25
3.2 BankService interface 26
3.3 Sketch of the BankAccount class 26
3.4 A CQL implementation of the BankService 28
3.5 newOperation methods. 31
3.6 Create a new operation with no arguments and a return value 32
3.7 Direct assignment of dependencies example 34
3.8 Scope creation example 34
3.9 Conditional Example 36
3.10 Loop example 37
3.11 Full transaction example 41
4.1 Future test. 54
4.2 Future test output. 54
4.3 ConsistencyLevelObject test. 55
4.4 ConsistencyLevelObject test output. 55
4.5 Scope test. 56
4.6 Scope test output. 57
4.7 Directly defined dependencies test. 58
4.8 Directly defined dependencies test output. 58
4.9 Condition test. 59
4.10 Condition test output. 59
4.11 Loop test. 60
4.12 Loop test output. 61
4.13 Creating bank account example. 64
4.14 Transfer transaction example. 64
4.15 Deposit transaction example. 65
4.16 Withdraw transaction example. 66
4.17 Balance verification transaction example. 66

Glossary

Linearizability In Linearizability the operations are executed atomically in order. Operations appear to take place instantaneously 9

Serializability In Serilizability a transaction schedule result is equal to the result of executing transactions serially (in order without overlapping in time) 9

Acronyms

ACID	Atomicity, Consistency, Isolation, and Durability 14
API	Application Programming Interface 4, 24, 25, 29, 30, 31, 33, 38, 40, 48, 49, 50, 57, 71, 72, 73
CRDTs	Conflict-free replicated data types 18
GPS	Global Positioning System 21
IP address	Internet Protocol address 30, 50
N/D	Not Defined 13
PoR	Partial Order-Restrictions 13, 15, 16
RAW	Read after write 43
REST	Representational State Transfer 4, 29, 48, 49, 72
TCC	Transactional Causal Consistency 18
WAR	Write after read 43
WAW	Write after write 43
YCSB	Yahoo! Cloud Serving Benchmark 52

Chapter 1

Introduction

This introduction provides a brief description of what will be covered in this thesis work and the goals set for the developed system. This Chapter provides insight into the motivation of the work realized. We proceed by describing the problems that motivated this thesis. And finally, we briefly describe the developed solution and provide insight on the contributions of this thesis work.

1.1 Context and Motivation

Modern systems require modern solutions. With the rise of highly influential applications and systems with heavy user usage and wide distribution, there is a need for consistent and fast support to meet the clients demands. To meet up to these requirements geo-replicated systems were developed. These systems are deployed closer to users, splitting latency across multiple servers and allowing for faster access and communication. However, splitting up a system across multiple services and locations raises problems with consistency and coordination. To fill this requirement, consistency and coordination control systems were developed to fit certain application and service necessities.

Distributed replicated database systems are often used as a way to obtain fault-tolerance and scalability. Most modern online services rely on geo-replicated data stores to manage the enormous amounts of data that are produced daily and to provide better geographic locality, availability, and disaster tolerance. In replicated systems data consistency is a crucial point. With data being spread throughout multiple servers and locations, if there is no control, inconsistencies are prone to happen. Thus, there is the need for the definition of a set of rules, that control data consistency between the distributed systems, to ensure that there are no irregularities in stored data that could cause disturbance to the applications and systems interacting with it.

Consistency usually requires a trade-off between latency, throughput and availability. In order for application developers to deal with this trade-off, several consistency levels have been defined along the years. Stronger consistency levels guarantee that only a consistent state can be observed across all replicas. Typically, this is ensured by forcing replicas to converge immediately, and wait upon every replica to confirm an update. This guarantees the same total order of operations across all replicas. However, protocols to ensure synchronization that provide strong consistency, need to form consensus between replicas and tend to be slower, less responsive and provide less availability. On the other hand, weaker consistency levels can provide higher availability and responsiveness but do not provide the same total order of operations across replicas. Many weaker consistency models only ensure that an update eventually becomes visible to all replicas, possibly resulting in replicas showing different states of the system to the same requests [1].

When developing distributed systems one must consider Eric Brewer's CAP theorem [2]. The theorem states that a distributed data store can only provide simultaneously two or less out of the following three guarantees: Consistency, Availability and Partition tolerance. These three CAP theorem guarantees might slightly diverge from the general notion for them. Consistency guarantee is the insurance that a read request receives the most recent write or an error; availability guarantee is the insurance that every request receives a non-error response, without the guarantee that it is the most recent; and partition tolerance guarantee is the insurance that the system operates continuously despite network partitioning. Distributed system networks can always fail, thus making it so that network partitioning must be tolerated, so developers must choose between consistency and availability. With consistency, upon partitioning the systems returns an error if data is not guaranteed to be up to date, whereas with availability the system processes the request and returns the most recent version of the data, even if there is no guarantee that it is up to date.

1.2 Problem

In this thesis we address a problem related to consistency coordination in replicated distributed systems, where developers have little control and a limited set of tools to set rules over their systems data consistency. There are three main approaches to defining consistency control in database systems:

- Consistency defined over data;
- Consistency defined over transactions;
- Consistency defined over operations.

Let us consider an hypothetical example of a database system that stores bank accounts and logs of transactions. Consistency defined over data ensures safety over data,

as the developer defines the consistency level for the data types. In the bank system example a bank account would have to be of stronger consistency level, as operations like a withdrawal must only happen over a consistent system. This approach causes operations that could be executed at weaker consistency levels to always execute at stronger consistency levels. As for example systems like Mixt [3], IPA [4], and Pileus [5] only support consistency defined over data.

With consistency defined over transactions the results are similar to consistency defined over data, as with this approach a consistency level is defined for the whole transaction. This results in operations insides the transaction that could execute at lower consistency levels, to execute at the higher consistency level of all the operations in the transaction.

The approach of using consistency defined over operations limits the data safety, as there is no known approach of using consistency defined over operations but using transactions for committing. Using only operations does not provide the safety and coherence of transactions.

We provide further analysis into these approaches in Chapter 2.

In light of the previously mentioned problems we ask ourselves: *Is it possible to provide the advantages of using both consistency over data and consistency over operations, and still commit using transactions?*

We aimed to find if it is possible to implement a system that could benefit from the safety of transactions but still execute operations with different consistency levels based on the developer choices and consistency level tagged data.

Additionally, we aim to provide a system that supports the consistency levels of the data store the developer chooses to use with the system and also, provide more, by controlling the non data store supported consistency levels in our system.

1.3 Developed Solution

During this thesis, the solution developed was a middleware system, called Ginger, for coordination of consistency levels in transactions that allows developers to attribute consistency levels to operations and data objects. Ginger system is composed by four main layers: a `front-end API` layer, a middleware layer, a `Data Service API` layer, and a back-end communication layer.

The developed system, Ginger, provides a set of functionalities for creating, processing and executing transactions. The system model focuses mainly on, attributing consistency to both data and operations, and allowing transactions to have multiple levels of consistency operations. Similarly to `MixT`, our system achieves this by breaking transactions into smaller ones, each bound to a particular consistency level. However, contrarily to `MixT`, this is done at runtime, not requiring a specific programming language. The transactions are processed in a two phase execution mode, where the first phase processes the

operations, attributing consistency levels to each operation and establishing dependencies between operations, and the second phase executes the operation in decreasing order of consistency level [3].

The `front-end API` provides a set of functionalities that allow users to create transactions and specify the consistency levels of the operations inside the transaction. Those functionalities include: creating operations with specific consistency levels, defining dependencies between operations, and communicating with the middleware for transaction execution. The Application Programming Interface (API) also processes the transaction code. The developers creates transactions and assigns the consistency to data and operations, using the API. It then processes this code searching for dependencies and defining the consistency levels the operations must execute at and in what order they must execute. Furthermore, it splits the operations by consistency levels and sends the sets of operations together to the middleware for execution.

The middleware is a system that can be implemented either attached to the API or as a stand-alone, communicating with the API via Representational State Transfer (REST). The middleware executes the processed transactions provided by the API. The way the middleware executes the transactions is based on the consistency level of the operations. It firstly executes the stronger consistency level operations and consequently executes operations in a decreasing order by their consistency level.

The `Data Service API` allows the developers to configure the services of the middleware. These services define the data store that will be used during a middleware session and what are the supported queries by the data store.

The back-end communication is part of the middleware and it establishes the connections between the middleware and the database systems. This part of the middleware commits the operations of the transaction to the database system. It also translates the responses of the database system into the expected result.

The current state of development of Ginger supports the Cassandra Database system and the linear and eventual consistency levels that this database system supports, but the end goal of the system is to support a higher variety of database systems and consistency levels. In the scope of this thesis the development of Ginger serves as a proof of concept for the approach taken on using consistency level defined both over operations and data in transactions.

1.4 Contributions

Firstly, we developed a model for annotation of consistency that provides ways of attributing consistency over data and operations simultaneously. This model allows for execution of transactions with multiple levels of consistency.

Secondly, we designed a middleware system for coordination of different consistency levels in transactions.

Thirdly, we developed a system, which is a prototype of the previously mentioned middleware, called Ginger, which allows for committing of transactions, with different levels of consistency for the operations of the transactions. Ginger provides tools for annotation of consistency levels over, data and operations, of a transaction. The system also allows for dependency control and both direct and indirect ways of establishing dependencies between operations.

Finally, we evaluated the system and provided insight into future developments of the system while also highlighting characteristics of our approach that we believe are research-worthy to the scientific community.

1.5 Document Outline

The remainder of this document is structured as follows. We analyze and provide insight, in Chapter 2, into the area of consistency coordination and existing systems with similarities to the system we developed. In Chapter 3, we start by presenting the characteristics of the Ginger system, followed by the means of usage of the system and finishing with an in-depth description of how the algorithm of Ginger processes and executes transactions. Consequently, in Chapter 4, we analyze the tests made to the system, which mainly focus on the functionally of the system and in proving that this approach to consistency coordination is a viable approach and worth exploring further. Finally, in Chapter 5 we provide some conclusions, based on the results and on the approach of development of our system, as well as some suggestion and plans for future work that we believe are either essential or would benefit the system greatly.

Chapter 2

State of the art

After the introduction of the goal of this project analyzed in the previous Chapter, this Chapter aims to cover the notions needed to understand the concept of the system and analyzes existing systems to provide an insight on the spectrum of guarantees to be covered by the system. Starting with a brief description of key concepts for consistency in distributed systems, followed by an in-depth analysis of the already existing systems, with similar characteristics to Ginger.

2.1 High Availability

When covering distributed system, one must have in mind high availability. Most current systems require high availability, as failures in the network may prevent database servers from communicating. The communication between the systems is slowed down by physical distance, network congestion and routing. Highly available system designs typically mitigate the effects of network partitions and latency, by allowing for closer to user servers and less communication between servers.

2.1.1 Requirements

To allow for high availability there must be an implementation of a system composed by multiple servers: a replicated distributed system. These systems usually have relaxed consistency, reducing the safety of system state visualization. The core requirement of high availability is that every user that can contact a server eventually receives a response from that server, even in the presence of arbitrary, indefinitely long network partitions between servers [6]. With this in mind, to provide high availability a system must relax consistency constrains or the communication between servers will cause a massive decrease in speed. This is due to the fact that if the system is ensuring strong consistency the

restraint and the wait time in communication between servers and the need to commit the transaction to every replica will pressure and lock the system, not allowing for high availability as users will have to wait for the system to be consistent.

2.1.2 Limitations

Highly available systems show limitations in consistency. As previously stated, to provide highly available systems, usually consistency requirements must be relaxed. This relaxation may result in inconsistency between replicas, as users may have different notions of the overall system when accessing different replicas. The inconsistency between replicas may sometimes need to be dealt with, as multiple states may interfere with each other.

2.1.3 System Requirements

To ensure high availability our system will make use of multiple consistencies. At the beginning causal consistency, eventual consistency and serializability will be the three supported consistencies. Eventual consistency as the weaker of the three allows for faster communication in high availability systems, as a transaction that has completed eventually becomes visible, this allows systems to not be constantly communicating and transactions to not need to be committed at every replica for a system to continue computing transactions.

2.2 Replication

A replicated system is a system that possesses two or more storage/computation devices that operate uniformly, similar to a unique system, but providing benefits such as, lower latency with the services being provisioned closer to users, data safety with multiple servers having copies of the same data, distributed processing and load balancing [1].

Replicated systems typically appear to work as would a non-replicated system. Upon a replica failure the system tries to continue providing service as if nothing happened. Replication is one of the core principles of high availability, as we previously stated in the High Availability Section 2.1.

Replication can be described as total or partial. In total replication all replicas contain the full set of data items in the system, whereas in partial replication each replica only holds a subset of the data items. Replication can also be described as either active or passive [1].

Active replication Active replication is the approach where all the servers represent equivalent roles and work as a group. All the servers independently and identically process the requests and reply. Upon replica failure if there are still replicas working the system remains working with no impact on performance, since the replicas that are up continue replying to requests normally. This system ensures sequential consistency as

all correct replicas process the same requests and in the same total order, resulting in all replicas being in the same state after a request [1].

Passive replication Passive replication is the approach where one server, so called primary replica, processes the request, if the request is an update the primary sends the updated state and the response to all secondary replicas, and then proceeds to respond to the client. Upon primary replica failure one of the other replicas takes place usually in a quorum voting. Meaning that upon primary replica failure, all replicas vote the primary replica by surpassing a minimum number of votes. The systems ensure Linearizability if the primary replica is correct as it sequences all the operations [1].

2.3 Consistency Levels

Consistency in database systems refers to the requirement that data written to the database must be in accordance with the rules defined: constraints, cascades and triggers. This ensures that any errors in programming cannot violate database constraints and that transaction correctness is responsibility of application-level code [7].

There are many ways of controlling consistency, usually described as consistency levels. These levels vary from strong to weak.

2.3.1 Strong Consistency

Strong Consistency is supported if all accesses are seen by all parallel processes in the same order. Therefor ensuring Serializability or Linearizability [8].

In this type of consistency, only one consistent state can be observed. Oppositely, in weak consistency, different parallel processes can perceive different states. These consistency may result in an indefinitely blockage in the operation under network partitions [9].

Sequential consistency Sequential consistency is a form of strong consistency and one of the strongest types of consistency. It requires every operation to acquire a mutual exclusion token [10]. The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

2.3.2 Eventual Consistency

In eventual consistency, convergence is expressed as eventual visibility. This is the concept that a operation **e** that is completed must eventually become visible to all sessions. This guarantee is assured by requiring that in each session, almost all operations that start after **e** returned must see **e** [9].

Eventual Consistency requires convergence of replicas, achieving consistency between replicas. In the absence of updates and failures the system converges towards a consistent

state. Updates may be reordered in any possible way and a consistent state is simply defined as all replicas being identical. Eventual Consistency is very vague in terms of concrete guarantees but is very popular for web-based services [11]. Eventual consistency can be satisfied as long as the client can reach at least one replica [9].

If eventual visibility is the only guarantee it may result in some anomalies. More guarantees can be introduced to eliminate those anomalies. These guarantees include, but are not limited to, the session and causality guarantees presented next.

2.3.2.1 Session Guarantees

When several operations are part of the same session, ideally those operations are expected to preserve the order at which they were issued. If a session terminates due to a failure scenario, a new session must be created, and the guarantees from the previous session are not valid. Session guarantees were introduced in the Bayou system [1, 12].

Read My Writes In eventual consistency it is possible that a user writes a message and when the user tries to read the message it might not show. This anomaly can be removed with *ReadMyWrites* guarantee. The use of *ReadMyWrites* guarantee ensures that for two events **A(read)** and **B(write)** if **A** happens after **B** then **A** has visibility over the changes made by **B**.

Monotonic Reads In eventual consistency it is possible for a user to read a certain state and a consequent read not show the same state has the previous read. This anomaly can be removed with *MonotonicReads* guarantee. The use of *MonotonicReads* guarantee ensures that for three events **A(read)**, **B(write)** and **C(read)**, if the event **A** sees the changes of event **B** then a consequent event **C** of **A** at least sees the state of changes of **B**.

Consistent Prefix In eventual consistency it is possible for a operation that writes **A** that was written before an operation that writes **B**, to be visible after such operation **B**. Since write **A** has a more recent timestamp, that causes **A** to appear before **B** in the returned value. This is potentially confusing since in the sequential semantics a post appends only at the end, and in this case, it shows the opposite. This anomaly can be removed with *ConsistentPrefix* guarantee. Using the guarantee *ConsistentPrefix* ensures that whenever we see an operation from a different session, we also see all operations that precede it in arbitration order, implying that the remote operations contained in each operation context form a contiguous prefix of the sequence of all operations (ordered by arbitration order) [13].

2.3.2.2 Causality Guarantee

Causality guarantee is the notion of happens-before for events that have a potentially causal relationship. The happened-before notion is a relation between the result of two

events, such that if one event happens before another event, the outcome must reflect that. Two operations of the same session may be causally related, has the program or the user may decide to issue an operation based on values returned by an earlier operation, and if an operation **A** is visible to an operation **B**, the value returned by **B** may depend on **A**.

Circular Causality If cycles are allowed in the happens-before order it may result in confusing results. Using the example stated in *Principles of Eventual Consistency* [13]: “The first user reads the wall and sees a single post saying “repeat me” (perhaps also including some promises about good things that will happen to you if you abide), and obeys by posting the same string. Then, it reads the wall and sees both the original post, and the repeated post. The second user goes through the exact same experience. Nothing seems wrong from the perspective of each individual user. However, something is definitely wrong: where did the original post come from? It spookily materialized out of thin air.” This is an example of circular causality. Using the guarantee *NoCircularCausality* ensures that there are no cycles in the happens-before order, preventing anomalies as the one stated above and circular causality.

Causal Arbitration Using the *CausalArbitration* guarantee ensures that if an operation **A** happens before an operation **B** then arbitration order is maintained in the happens-before order. Practically, causal arbitration can be ensured by obtaining timestamps from well synchronized clocks or from a single arbitration node [13]. We will further discuss the role of clocks in Section 2.7.

Causal Visibility If the happens-before order does not enforce visibility, it may result in a confusion where a certain user wont correctly understand the situation as a result of not having the entire information on the situation. Using the *CausalVisibility* guarantee ensures that if an operation **A** happens before an operation **B** then **A** is visible by **B**.

2.3.3 Causal Consistency

Causal consistency implies that if two operations **o1** and **o2** from the same session are applied to two different replicas **r1** and **r2**, the second operation cannot be discharged until the effect of **o1** is included in **r2**. Causally consistent operations are required to see a causally consistent snapshot of the object state [9].

Causal consistency disallows some of the anomalies of the processing of asynchronous operations and ensures that message propagation between replicas is causal. As stated in [10] if a replica sends a message containing the effect of an operation **o2** after it sends or receives a message containing the effect of an operation **o1**, then no replica will receive the message about **o2** before it receives the one about **o1**. In this case we say that the invocation of **o2** causally depends on that of **o1**.

2.3.4 Explicit Consistency

In Explicit Consistency [14] programmers define the application-specific correctness rules that should be met. These rules are expressed as invariants over the database state. Using the example written by the authors [14]: Even if each replica maintains some invariant locally, concurrent updates might still cause violation. Consider for instance a tournament with a maximum capacity, limiting the cardinality of the set of enrolled players. Two replicas could concurrently enroll players into the same tournament, each one respecting the capacity. However, if the merge function is the union of the two sets of players, the capacity might be exceeded, nonetheless.

A system that supports Explicit Consistency identifies which operations would be unsafe under concurrent execution and allows programmers to select either violation-avoidance or invariant-repair techniques. In invariant repair operations can execute concurrently and their outputs are merged and include code to repair the invariants. Using another example of the authors [14] in a graph data structure that supports add and remove operations, the addition of an edge in a replica and the removal of a vertice from that edge in another replica, may result in the merged state ignoring the hanging edge to ensure the invariant that an edge connects two vertices. Violation avoidance technique consists on the restriction of concurrency to avoid the invariant violation.

2.3.5 Hybrid Consistency

Hybrid consistency models allow the programmer to request stronger consistency for certain operations and weaker consistency for other operations. Thereby, introducing synchronization and performance in replicated systems. A Hybrid model can execute some operations under some consistency level and others under a different consistency level. Although this model allows for fine-tuning and high control it is a complex model to use.

Balancing consistency levels is an important part of hybrid models. The usage of strong consistency in too many operations may negatively impact performance and availability, but, on the other hand, using it in too few operations may impact correctness. To achieve balance the programmer must take into consideration which anomalies are disallowed by a particular consistency strengthening and whether disallowing these anomalies is enough to ensure correctness [10].

RedBlue Consistency RedBlue Consistency model is a hybrid model that classifies operations as either red or blue. Red operations are guaranteed sequential consistency, and blue operations causal consistency [10].

Table 2.1: Works addressing multiple consistency levels

Name	Consistency over	Consistency Defining	Consistency levels	Granularity	Dependencies	Implemented over
[3] MixT	Data	Static	Linearizable, Causal, Eventual	Transaction	Explicit	Postgres
[15] Gemini	Operation	Static	RedBlue	Operation	Explicit	MySQL
[9] Quelea	Operation	Static	Eventual, Causal, Strong	Operation	Explicit	Cassandra
[16] Consistency Rationing	Data	Dynamic/Static	Session, Serializable, Adaptive (between previous 2)	Operation	Implicit	Amazon's Simple Storage Service
[17] SIEVE	Operation	Dynamic/Static	RedBlue	Operation	Explicit	MySQL, Gemini
[14] Indigo	Operation	Static	Explicit	Operation	Explicit	SwiftCloud
[4] IPA	Data	Dynamic/Static	Strong, Eventual	Operation	Explicit	Cassandra
[18] Olisipo	Operation	Static	Partial Order-Restrictions (PoR)	Operation	Explicit	MySQL, Gemini, SIEVE
[19] Homeostasis protocol	Transaction	Dynamic	Inconsistent into Strong	Transaction	Explicit	MySQL InnoDB Engine
[20] Walter	Transaction	Static	Eventual, Serializable	Transaction	Implicit	Not Defined (N/D)
[5] Pileus	Data	Dynamic/Static	Strong, Eventual	Operation	Implicit	N/D
Our system	Data/Operations	Static	Linearizable, Eventual, Causal	Transactions	Implicit/Explicit	Currently: Apache Cassandra [21]

2.4 Multiple Consistency Levels

Some systems make use of only one stronger consistency level, but using only one consistency level can be slow as the operations of a transaction are committed at the consistency level required by the most sensitive operation, causing delay and slow committing for operations that could execute at a weaker consistency. The usage of multiple consistency levels can solve this problem allowing for faster committing of operations that do not require stronger consistencies, and safety for the operations that do require stronger consistencies.

In this Section and in Table 2.1 we analyze and compare some of the systems that use multiple consistency levels, comparing the different approaches and functionalities provided by such systems. We analyze the systems with the goal of defining the foundation for the development of our system.

2.4.1 Consistency Over and Granularity

In this Subsection we analyze two attributes that can be easily confused but while partially overlapping are quite different. Consistency can be defined on transactions, operations or data whilst Granularity is assumed at the transaction/operation level. The difference is, even though consistency is defined for transactions, operations or data by attributing a consistency level or a certain set of rules to one of these attributes, the systems have a granularity level in which they define where to lock and define consistency. A system may label a level of consistency to a data item but lock at transaction level making it so that all changes made by that transaction are seen after it ends.

Consistency defined on data allows to handle the data according to its importance. However, transactions and operations may see different consistency levels due to the accessing of different data. If a single transaction processes data from different categories,

every record touched in a transaction is handled according to the category guarantees of the record [16].

In systems like MixT [3] consistency is treated as a form of data integrity, expressing it with labels on types in the language. To ensure consistency guarantees, weaker consistency information should avoid influencing stronger consistency information. On the other hand, in systems like Gemini [15] consistency is defined on operations, as these systems mark operations for either a weaker consistency (blue) or a stronger consistency (red). Some systems, such as Walter [20], opt for defining consistency over transactions, relieving developers from Atomicity, Consistency, Isolation, and Durability (ACID) concerns and being a easier development approach.

In the context of multiple consistency levels, granularity can be coarse-grained, defined by locking at the transaction level. In transaction level locking, results of the changes made by the transactions are seen at the end of the execution, and it does not allow for other transactions to make changes to data that the transaction is accessing, reducing concurrency greatly. Locking at the transaction level results in a more secure way of locking, as it usually assumes the stronger consistency level between all the operations of the transaction for all operations.

Oppositely, granularity can be fine-grained, defined by locking at the operation level. Single operations or multiple operations can be locked together, allowing for a more adapted and controlled consistency, but resulting in a bigger overhead, necessity of control and harder implementation. Operations may be committed at different consistency levels. As an example, Indigo [14] uses this approach as the developers provide the invariants, or consistency rules, that the system must maintain and the systems analyses which operations can be safely executed without coordination and, secondly, provide an alternative set by the developer, for the remaining operations.

Our system allows for consistency to be defined over data and operations. We will further clarify on this matter in Chapter 3. Granularity wise we lock at the transaction level, but we allow for transaction splitting. In Section 3.5 the matter of transaction splitting will be revised.

2.4.2 Consistency Defining

The process of defining consistency can be static or dynamic. Defining consistency statically is the attribution of a consistency level or guarantee to one of the three attributes, mentioned above in Subsection 2.4.1, in anticipation to the execution of the transactions/-operations. System MixT [3] achieves this as developers attribute consistency level/guarantee labels to data types.

In dynamically defined consistency the system does it at run time, adapting consistency levels based on evaluations or defined policies, made with the goal of optimizing performance without losing consistency.

A good example of the usage of both static and dynamic defining is Consistency

Rationing [16]. In this system consistency is statically defined as data is divided into three categories, A, B and C that, respectively, ensure strong, adaptive and session consistency. Dynamically defined consistency is used in the system as data in the B category is handled with adaptive consistency where the system dynamically calculates the probability of conflicts and ensures either strong or session consistency.

The goal of our system is to support both static and dynamic consistency defining. The current state of Ginger only supports static consistency definition.

2.4.3 Consistency Levels

The goal of our system is to support multiple consistency levels but the prototype developed during the work of this thesis only supports linear and eventual consistencies. The consistency levels of the systems compared above in Table 2.1 are mostly covered in Section 2.3, but the following concepts should be taken into consideration:

Shadow operations Systems, Gemini [15] ,SIEVE [17] and Olisipo [18], use shadow operations. The idea behind shadow operations is splitting each application operation in two:

- a **generator operation** that identifies the changes the original operation should make but with no side effects. This operation is executed only at the primary site against some system state;
- a **shadow operation**, produced by the execution of the generator operation, which is executed at every site.

This solution allows the generation of state-specific shadow operations with different properties, which can then be assigned different colors in the RedBlue consistency model.

Partial Order-Restrictions PoR consistency is used by the system Olisipo [18], and it is composed by three components:

1. A set of restrictions, which specify the visibility relations between pairs of operations;
2. A restricted partial order (or short, R-order), which establishes a (global) partial order of operations respecting operation visibility relations;
3. A set of site-specific causal serializations, which correspond to total orders in which the operations are locally applied.

The Olisipo [18] system is designed by the authors of both Gemini [15] and SIEVE [17] and is based on the previous two developments. The introduction of PoR consistency has the goal of overcoming the RedBlue consistency limitations allowing for a fine-grained approach as developers reason over a set of restrictions imposed over admissible partial

orders across the operations of a replicated system. The RedBlue showed limitations in terms of a high overhead in geo-distributed settings, labeling shadow operations as strongly consistent when there were no need for such, therefore the introduction of PoR allowing for flexibility at the level of coordination.

2.4.4 Dependencies

The dependencies between operations or transactions can either be set explicitly with the system or the developer defining rules or orders for them, or they can be set implicitly as the consistency defined by the systems ends up being enough to ensure the order between operations or transactions.

Olisipo [18] system has a very direct approach to explicitly defined dependencies as a set of restrictions is set, which defines relations between pairs of operations and those relations form a restricted partial order that consequently establishes an order of operations respecting operation visibility relations.

Out system, Ginger, supports dependencies both implicitly and explicitly. As we will discuss in Section 3.4

2.4.5 Implemented Over

The implementation of the systems in Table 2.1 show similarities, as most of them are written in one or more programming languages over a database system. Some systems are also implemented in a programming language over other systems, as does SIEVE [17] by being implemented in Java over Gemini.

The goal of our system is to support multiple database systems but we used only Apache Cassandra Database [21] during the development stage as we discuss later in Section 3.7.

2.4.6 Evaluation

When it comes to evaluation most systems we analyzed in this Chapter [5, 14–20] tend to have a very similar approach, consisting in building their systems over a Database system. These systems use and possibly modify existing benchmarks to evaluate, how the systems performs in relation to the contributions they provide, how the systems performs in "real-word" environment and how it stacks up against other systems. Systems [15, 19, 20] also develop a micro-benchmark for a faster and focused evaluation on the principal contributions of the system.

Common used benchmarks include Rubis and TPC-W:

- Rubis - is the Rice University Bidding System [22], an eBay-like auction site. RUBiS implements a large number of operations and transactions requiring varying levels of consistency and isolation;

- TPC-W - TPC Benchmark W [23] is a transactional web benchmark. Is a simulation of a controlled Internet commerce business oriented transactional web server. TPC-W handles scalability by establishing a relationship between the number of concurrent sessions and the size of the store, measured in terms of the number of items in the inventory.

Some systems [14] also develop specific applications allowing the evaluation to focus on the crucial contributions of the system.

MixT [3] could not use exiting benchmarks as the authors could not find existing benchmarks for mixed-consistency transactional systems. So, they developed two new benchmarks to evaluate the mixed-consistency. One micro-benchmark without mixed-consistency transactions and an extension of the previous one with mixed-consistency transactions. To fit our system we also chose to develop our own benchmark.

2.5 Ensuring Causality

Causal consistency is attractive for high availability, avoiding high latencies, partition-intolerance associated with strong consistency and some possible anomalies with eventual consistency. Ensuring causality can take many paths. We analyzed four state-of-the-art approaches on which to base our implementation. Those designs are the following: Cops [24], Cure [25], GentleRain [26] and SATURN [27]. Follows a brief description of the systems.

Cops Cops [24] (Clusters of Order-Preserving Servers) is a distributed storage system that provides causal+ consistency. Causal+ consistency is the strongest model compatible with availability for individual operations, since it ensures that the causal ordering of operations is respected. The system is a key-value storage to run in a small number of Data Centers, with each Data Center having a local COPS cluster with a complete replica of the data.

The clusters are set up with linearizability providing possible scalability by partitioning the key space into linearizable partitions and having clients access each partition independently. This ensures the whole systems remains linearizable. Replication between clusters is asynchronous ensuring low latency for client operations and availability [24].

GentleRain GentleRain [26] is a protocol of periodic aggregation that determines whether updates can be made visible in conformance with causal consistency. The protocol makes usage of loosely synchronized physical clocks to attach scalar timestamps, to ensure causal consistency. The usage of a single scalar timestamp allows for reduced storage and communication overhead. The physical clock value used is the physical clock value of the server where they originate. These timestamps provide a total order on all updates, consistent with the causal order of events. Updates are visible only if they do not violate causal

consistency. Local updates are always immediately visible whereby nonlocal updates are visible when their update timestamp is smaller than the global stable time.

GentleRain improves in comparison to other causally consistent key-value stores by eliminating dependency check messages for updates and using only a single physical timestamp to track dependencies. Therefore distinguish it from systems such as COPS [24] which explicitly tracks individual dependencies.

The elimination of dependency check messages improves throughput in comparison to other systems. The usage of a single timestamp allows for a concise representation of dependencies and reduce storage and communication overheads. Even though GentleRain achieves better throughput and reduces storage and communication overhead, it incurs longer latencies in making updates remotely visible [26].

Cure Cure [25] introduces a consistency model, Transactional Causal Consistency (TCC) model that extends the causal+ consistency with interactive transactions and Conflict-free replicated data types (CRDTs) for replica convergence.

The interactive transactions provide the possibility to combine read and write operations flexibly in the same transaction ensuring that a read represents a view of the data store that includes the effects of all transactions that causally precede it because it is read from a causally consistent snapshot, and that transactions respect atomicity as all updates either occur and are made visible simultaneously, or none of them does. CRDTs are high-level data types that can be replicated and modified concurrently while guaranteeing that replicas converge [26]. CRDTs allow for a better interface for programmers than the key-value interface.

Cure keeps multiple versions of each object to supply the requests from causally consistent snapshots. Each version is stored with metadata to encode its causal dependencies. Cure uses a vector clock to annotate its updates with the commit time that produces a partial order respecting causal consistency.

Summing it up cure supports causal+ consistency so that no update happens before another, CRDTs that provide intuitive semantics and guaranteed convergence and transactions, ensuring that multiple objects are both read and written consistently [25].

SATURN SATURN [27] uses metadata to control update visibility between Data Centers. SATURN decouples data and metadata management. It relieves the Data Store from managing consistency across Data Centers. The decoupling allows SATURN to handle heavier loads independent of data management size. SATURN management is based on labels that uniquely identify operations and have fixed size. The generation of labels is responsibility of the Data Centers, and they pass them to SATURN in an causality consistent order, attached to the corresponding update payload. The system propagates those labels among Data Centers in causal order, and the Data centers then apply remote updates locally when they have received both the update payload, and the label. In SATURN the labels can be compared and ordered globally, being so that the total order

defined by them respects causality. The system enables genuine partial replication by selectively delivering labels to the set of interested Data Centers.

Causal order is a partial order, which means that there can be, multiple serializations of the labels that respect causality. This property allows SATURN to provide to each Data Center specific performance maximizing label serialization. SATURN allows Data Centers to apply updates in an order set by SATURN, to each Data Center, different from the global order of timestamps. This SATURN defined order respects causality. There are two different types of labels in SATURN:

- An update label that is generated when a client issues a write request.
- A migration label that is created when a client needs to migrate to another Data Center

Summing up, SATURN decouples data and metadata management, implementing labels with constant size in a decentralized way, allowing it to handle heavier loads [27].

Comparison Looking at the solutions previously described, GentleRain implements a coarse-grained approach, tracking by compressing metadata into a single scalar. Oppositely Cure goes for a more fine-grained approach, relying on a vector clock with an entry per Data Center. The metadata management provides a low visibility latency penalty but severely penalizes the throughput due to the computation and storage overhead.

On the one hand, GentleRain induces low penalty on throughput but puts a heavy weight on the visibility latency. This is due to the large amount of false dependencies inevitably introduced when compressing metadata. But, on the other hand, Cure exhibits low visibility latency penalty but due to the metadata management is punished in the throughput associated with the computation and storage overhead.

SATURN eliminates the trade-off between throughput and data freshness in the previous solutions. It does it so by having small and constant metadata size. The system also ensures a close to weak-consistent systems visibility latency of updates with metadata propagation techniques.

2.6 Controlling Eventual Consistency

Eric Brewer's CAP theorem [2], as stated in Section 1.1 is based on the fact that one cannot have the three properties: Consistency, Availability and Partition Tolerance at the same time, and eventual consistency is a clear example of this, as it trades Consistency to obtain the other two properties.

Looking at two properties of distributed systems: Safety and Liveness. Safety is the guarantee that nothing bad happens, whereas liveness is the guarantee that something good eventually happens. The problem with eventual consistency is that it does not provide safety and only provides liveness. Safety may be broken due to the fact that in

eventual consistency at any given time there is the possibility of inconsistent behavior, for example the system can return any data and be eventually consistent, as it might converge later. The only guarantee is that eventually the system will converge in the future and it will be consistent [28]. Eventual consistency is a minimum requirement for data consistency, as it is already very difficult to reason about. Latency benefits a lot from eventual consistency as consensus between the servers can be delayed and not immediately perform updates on all replicas.

2.6.1 Implementation

Eventual consistency is mostly straightforward to implement. Replicas must send information between each other on the writes. A server can, upon a write request, send this information to all servers in the cluster, but it must be aware.

A server cannot be waiting on all the servers responses to acknowledge the local write, because if at least one server is down or partitioned from the cluster the request will hold endlessly. A solution is to make the servers send the requests in an asynchronous way, and all operations update locally, providing reduced latency. This approach, with immediate update on local request, can break data consistency and cause problems. A solution stated in *Eventual Consistency Today: Limitations, Extensions, and Beyond* [29] provides a good middle ground between consistency and availability that is to return the update after the write has been acknowledge by N replicas, allowing it to survive N-1 failures of replicas.

2.7 Ordering Events

Regardless of the type of consistency guarantees that the systems choose to provide, there is still the need to keep track of the relative order of events to ensure that the state of the data store stays consistent and updates are observed in the expected order. The way to track this ordering correctly is to use some type of clock to timestamp the events.

2.7.1 Clocks

There are different properties that characterize clocks resulting in the existence of a wide variety of clocks:

- Scalar Clocks - Clocks that maintains a single value;
- Vector Clocks - Clocks that maintains a value per node in the system;
- Matrix Clocks - Clocks that maintains a value per link between nodes in the system.

Scalar clocks use less space than vector or matrix clocks but are generally impossible to know if two events are concurrent or causal.

Independent to the number of values in the clock, they are also characterized if they capture values as the real time instant of the event occurrence, known as physical clocks or if they use logical values that count relevant events occurrences observed, known as logical clocks. There is also a less used approach, but still relevant, that is hybrid clocks, that use both a physical clock value and a logical value. Physical clocks tend to be better than logical clocks. This is due to the fact that physical clocks allow, to keep track of causal relations and to relate real time timestamps with the events. Opposed to this advantage physical clocks suffer from differences between clocks in different nodes of the system, not allowing them to be perfectly synchronized. One possibility to synchronize physical clocks is the usage of distributed clock synchronization algorithms, but these may still result in differences between clocks with variations of hundreds of milliseconds. Another possibility is to use appropriate hardware such as Global Positioning System (GPS), but these tend to be costly and not common in public cloud infrastructures.

Weak consistencies and clocks Taking a look at common approaches of systems with weaker consistency models such as causal consistency, these make use of a wide variety of clocks, for example COPS [24] uses scalar clocks and clients keep the last clock value of all objects read in the causal past representing their causal history, whereas GentleRain [26] uses physical clocks for timestamp updates, detecting conflicting versions, creating read snapshot time for read-only transactions, and computing the global stable clock.

Strong consistencies and clocks Taking a look at common approaches of systems with stronger consistency models, these also make use of a wide variety of clocks, for example Clock-SI [30] uses loosely synchronized physical clocks to identify each transaction with a read timestamp, providing a readable consistency snapshot. A very different example is Spanner [31] that uses physical clocks but assigns a time interval upon a write, where it is considered unsafe to read that transaction, blocking this possibility.

2.8 Conclusion

In conclusion, there is a wide variety of systems and approaches available with a lot of different attributes and ways of attacking the problem of controlling consistency in data systems. In Ginger we chose the approaches which we thought would benefit the developers the most. Ginger innovates by allowing for consistency to be defined to both data and operations whilst using granularity over transactions. We chose this approach, as after analyzing it, seemed to provide the benefits of both the consistency over data approach and the consistency over operations approach. This approach and its results are further analyzed in Chapters 3 and 4, respectively.

Chapter 3

Ginger

In this Chapter we approach the Ginger middleware system, starting with an overview of the system in Section 3.1, following with an analysis of the technical architecture in Section 3.2. Subsequently, we provide an in-depth description of its features and components: the `Data Service API`, the `front-end API`, the processing of transactions, the middleware system and the back-end communication, in Sections 3.3, 3.4, 3.5, 3.6 and 3.7 respectively.

3.1 Overview

Ginger is a middleware, that acts as a coordinator between developer application operations and one or more database systems. The system provides functionalities to coordinate the consistency of data and operations maintaining the dependencies between the two.

The system is written in Java programming language. Ginger innovates by providing consistency coordination both on the data and on the operations. In Ginger middleware, the consistency can be defined over data, operations or both at the same time, providing the developer with another level of control over the consistency of their system. The middleware achieves this by analyzing code written by the programmer with annotations and defining consistency levels for each operation. Ginger uses transaction and breaks them into smaller transactions based on operation consistency levels and the dependencies between operations. It processes transactions ahead of the execution to define the dependencies and the consistency level of the operations.

The framework provides a set of interfaces allowing developers to define the database system to use and how it should behave for the chosen consistency levels. In Ginger the developer can either use annotations, to define transactions and operations, or directly

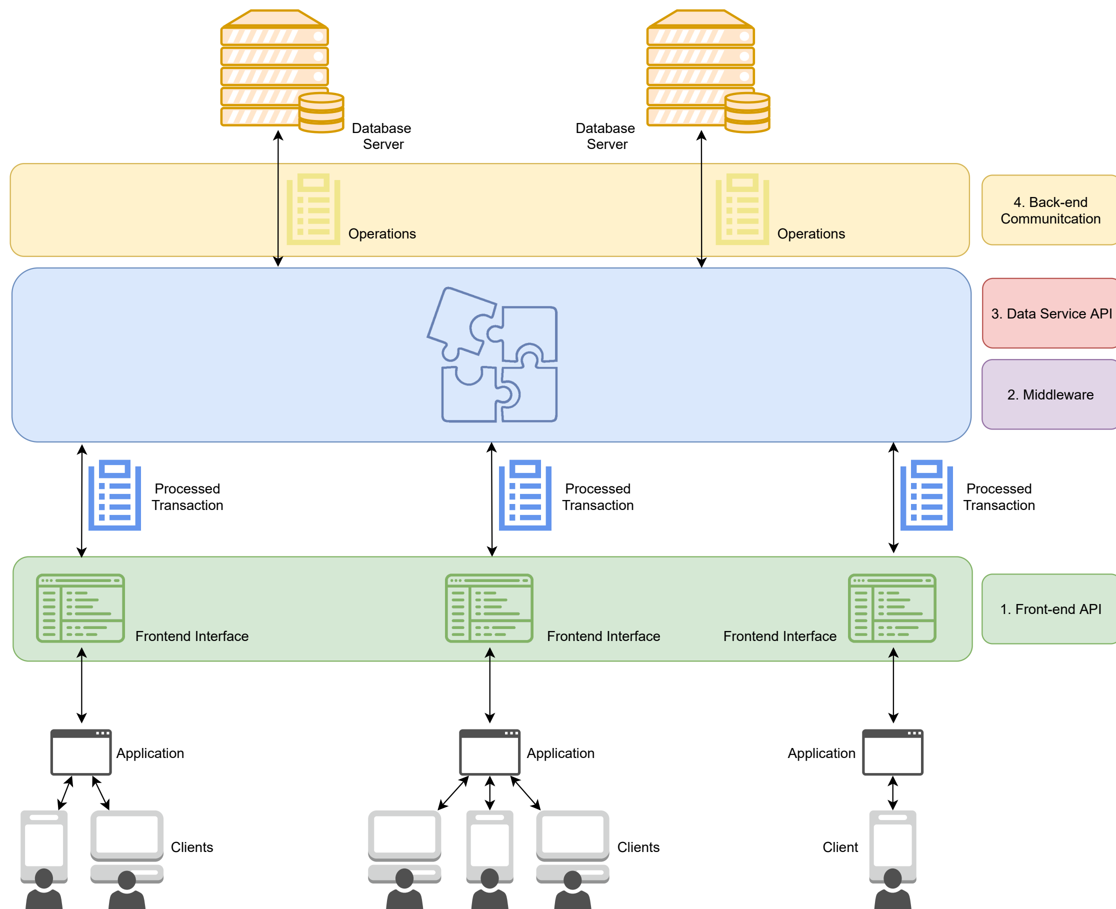

Figure 3.1: Diagram representative of a simplified version of the system architecture using only 3 applications, 2 Database Servers and 1 instance of the middleware.

specify the operations and the consistency levels of both operations and data, using the Ginger API functionalities. This system has the goal of supporting multiple database systems and multiple consistency levels.

3.2 System Architecture

The architecture of the Ginger solution, as represented in Figure 3.1, is composed by four layers.

The `front-end API` (1. in Figure 3.1) provides a library that offers a user-friendly approach,where the developer defines the transactions and operations with either annotations or by calling API methods. The front-end processes the system transactions, separating and defining the dependencies between operations and furthermore provides the middleware with the processed transactions for execution.

The middleware (2. in Figure 3.1), is a working software that can be deployed

```
public interface DataService {
    QueryLanguage getQueryLanguage();
    String getNameSpace();
}
```

Listing 3.1: The DataService interface

anywhere and communicates with both the developers system front-end API and the databases. This component receives the processed transactions and executes them according to the consistency levels defined by the developer and the dependencies between data and operations. The middleware commits the transaction's operations to the data stores. Multiple instances of the middleware can be deployed together, with at least 1 database server for each instance.

The Data Service API (3. in Figure 3.1) provides a library for storage system configuration. The developer can define through the API, the storage system it will use and what type of queries it supports.

The last component is the back-end communication (4. in Figure 3.1), which is in charge of communicating with the database system and committing operations to the database.

3.3 Data Service

Ginger presents the combination of a data store with a set of operations (queries), that can be performed over such store, as a Data Service.

The Data Service API provides the means for programmers to define, implement and register Data Services.

These services are ruled by the DataService interface (Listing 3.1) that defines two methods: getQueryLanguage that conveys which query language is used to perform the queries, and getNameSpace that allows for the definition of a namespace for the target data, such as a key space in distributed NoSQL data stores. The QueryLanguage type defines an enumeration over the query languages known to Ginger.

3.3.1 Service Definition

A Data Service specifies the set of operations that the service allows over its data store. The specification is given as an interface that extends DataService, being the signature of each operation of the form:

```
Query<T> operationName(Object... arguments);
```

```
1 public interface BankService extends DataService {
2     Query<Void> insertBankAccount(Object... opData);
3     Query<Void> addLog(Object... opData);
4     Query<Void> deleteBankAccount(Object... opData);
5     Query<Void> deposit(Object... opData);
6     Query<Void> withdraw(Object... opData);
7     Query<Double> getBalance(Object... opData);
8     Query<List<Integer>> getAllAccountNumbers(Object... opData);
9 }
```

Listing 3.2: BankService interface

```
1 public class BankAccount implements LinearConsistency {
2     private long accountNumber;
3     [...] // omitted
4     public BankAccount(long accountNumber)
5     public long getNumber() { return accountNumber; }
6 }
```

Listing 3.3: Sketch of the BankAccount class

that obliges the reception of an array of generic objects, and the return of a query, instance of Query<T>, where Query<T> represents a Ginger query that returns a result of type T.

This strict specification enables the representation of operations as functional objects(implementations of Java's Function interface), necessary for further usage of the API, as we will detail in Subsection 3.4.3.

Java's Function interface

Function<T,R> is a functional interface that can be used as a lambda expression or method reference. T denotes the type of the function's only argument and R denotes the function's return type.

Listing 3.2 showcases the definition of an interface for a banking Data Service.

The bank accounts are represented by instances of class BankAccount (Listing 3.3), which comprises the account's data, such as number, owner, among others. This class implements the LinearConsistency interface which extends the ConsistencyLevelObject interface. We will further explain the need for this in Subsection 3.4.3.

3.3.2 Service Implementation

A service implementation (an implementation of the service's interface) must define how each service operation is translated into a query to the target data store. Listing 3.4 exemplifies how some operations of the `BankService` may be implemented to perform queries on a Cassandra database, by using the `Cassandra Query Language` (`CQL`).

This example, besides the omitted operation which are non essential to comprehension, contains: methods for the creation of queries for the deposit and withdrawal of money from a bank account, which do not return a result, and a method for the creation of a query to verify the balance of a bank account, which returns a `Double` value representative of the account balance. Upon creation, a `Query` object requires an argument with the query to the data store in the query language defined, such as `CQL`. When targeting Cassandra databases, the query may be given either as a string or an object of type `BuiltStatement`, from the `Cassandra Datastax API`. Optionally, `Query` also accepts a second argument, of type `Function`. This function has the goal of translating the response from the database into the expected result. Using the `getBalance` operation as an example, the query for the balance in the database returns a `ResultSet` with the row containing the balance value of bank account that matches the account number provided. The given translation function (line 26) then processes the `ResultSet` to return the value of the balance as a double.

3.3.3 Service Registration

In order to be available for use in the developers' API, a `Data Service` must be previously registered in Ginger. This registry requires four values:

- The `Data Service` **interface** that specifies the service's operations.
- A **key** to identify the service in the space of the given `Data Service`. The pair (service interface, key) defines a system-wide unique identifier that may be used in the client's API to refer to any particular service instance.
- The service's interface **implementation**.
- A `DataStore` object that conveys information about the **data store** holding the service's data. The pair (service implementation, data store) represents a `Deployed Data Service`, i.e. a ready to use `Data Service`.

The `DataStore` class defines the means for Ginger to interact with a given data store. Besides the target data store's location (host name and port), the list of supported query languages, and the list of supported consistency levels, it requires the definition of the following two entities: a `Data Store Session` that defines how to create a session with the data store, and a `Query Executor` that defines how to execute a given query with a given consistency level in a data store session.

```
1   public class CassandraBankService implements BankService {
2       private final String table = "bankAccounts";
3       [...] // omitted
4       public Query<Void> deposit(Object... opData) {
5           BankAccount account = (BankAccount) opData[0];
6           double accountBalance = (double) opData[1];
7           double depositValue = (double) opData[2];
8           Update update = QueryBuilder.update(table);
9           update.where(QueryBuilder.eq("accountNumber", account.getNumber()));
10          update.with(QueryBuilder.set("balance", accountBalance + depositValue));
11          return new Query<Void>(update);
12      }
13      public Query<Void> withdraw(Object... opData) {
14          BankAccount account = (BankAccount) opData[0];
15          double accountBalance = (double) opData[1];
16          double withdrawValue = (double) opData[2];
17          Update update = QueryBuilder.update(table);
18          update.where(QueryBuilder.eq("accountNumber", account.getNumber()));
19          update.with(QueryBuilder.set("balance", accountBalance -
             ↪ withdrawValue));
20          return new Query<Void>(update);
21      }
22      public Query<Double> getBalance(Object... opData) {
23          BankAccount account = (BankAccount) opData[0];
24          Select select = QueryBuilder.select("balance").from(table);
25          select.where(QueryBuilder.eq("accountNumber", account.getNumber()));
26          return new Query<Double>(select, (ResultSet r) ->
             ↪ r.one().getDouble("balance"));
27      }
28      [...] // omitted
29      public QueryLanguage getQueryLanguage() {
30          return QueryLanguage.CQL;
31      }
32      public String getNameSpace() {
33          return "Bank";
34      }
35  }
```

Listing 3.4: A CQL implementation of the BankService

Although Ginger has the goal of supporting multiple data store types, currently only Cassandra database is supported. This supports the CQL query language, and the LINEAR and EVENTUAL consistency levels. A Cassandra database system is partitioned in key spaces that comprise the tables of data. In this context, the service name space is used to define key space to use. The registry of the CassandraBankService with key "Cassandra Ginger Bank"supported by a Cassandra database located at "SomeHost", port 9042, is done as follows:

```
1 Registry.addService(
2     BankService.class, // the Data Service interface
3     "Cassandra Ginger Bank", // the key
4     new BankCassandraService(), // The Data Service implementation
5     new CassandraDataStore("SomeHost", 9042)); // the data store
```

3.4 Front-end API

The front-end API consists of a group of functionalities that allow programmers to attribute consistency to data and operations, and manage the consistency levels and dependencies of the operations of the transactions. The API provides the developers with a set of interfaces and methods that simplify the execution of transactions with a variety of functionalities for control of constraints, including dependencies between operations, different consistency levels and preserving consistency between accesses to the same portions of data. The API processes the transactions written by the programmer and communicates directly with the middleware layer, which is in charge of committing the transactions to the data store.

3.4.1 Ginger Session

The interaction with a Ginger Data Service can only be done in the context of a session. Currently, two kinds of sessions are available: a *local* one, that uses the Ginger middleware as a local library, and a *remote* one, that enables the interaction with a remotely deployed middleware layer, via REST.

To unequivocally identify the Deployed Data Service, both require the Data Service interface and the registry key. An example of the creation of a local session follows:

```
1 Session<BankService> session = new LocalSession<BankService>(
2     BankService.class, // the Data Service interface
3     "Cassandra Ginger Bank"); // the key
```

To obtain a session from a middleware system that is detached from the API, the creation of a RESTSession in necessary. This creates a REST connection service between

the API and the middleware. To that end, the programmer must complement the arguments given to LocalSession with the Internet Protocol address (IP address) and port of communication of the middleware:

```
1  Session<BankService> session = new RESTSession<BankService>(
2      BankService.class, "Cassandra Ginger Bank", "85.222.44.6", 9760);
```

3.4.2 Transaction

Ginger is a transactional system that hence makes use of transactions to manipulate the data stored in the target data store. The creation of a transaction is done by creating a concrete implementation of the Transaction abstract class, instantiated with the transaction's behaviour in method code. This method is what allows the developer to create new operations and define the dependencies between operations and their consistency levels:

```
1  Transaction t = new Transaction() {
2      public void code() {
3      [...] // omitted - code of the transaction
4      };
5  };
```

This way the developer creates a new transaction that will further be processed and delivered to the middleware for execution, which we will further explore in Subsection 3.5. The class Transaction provides the methods for the creation of operation and control of the transaction. The code of a transaction can be divided into two categories: the one that executes locally, and the one that interacts with a data store. The latter must be expressed in the form of Ginger operations.

3.4.3 Operation

A database transaction is a sequence of operations that perform work in a certain database, in an isolated manner from other transactions. Creating a new operation in Ginger requires the developer to call the newOperation method inside the Transaction code method referenced in previous Subsection. The new operation embeds our approach of allowing consistency levels to be defined over data or operations. In what concerns the former, the programmer may indicate a ConsistencyLevelObject that defines the consistency level of the operation. As illustrated in Listing 3.5, lines 6 to 9, this information is passed as the second argument to the newOperation method. Regarding consistency defined in operations, the programmer must explicitly indicate such consistency level in the first argument of the newOperation method (Listing 3.5, lines 1 to 4).

```
<T> Result<T> newOperation(
    ConsistencyLevel consistencylevel, // the consistency level
    ServiceOperation<T> operation, // the method reference to the  operation
    Object... opargs); // the operation's arguments

<T> Result<T> newOperation(
    ServiceOperation<T> operation, // the method reference to the  operation
    ConsistencyLevelObject object, //Consistency level defining object
    Object... opargs); // the operation's arguments
```

Listing 3.5: newOperation methods.

The `ConsistencyLevel` class is an enumerator object, which allows the developer to select one consistency level from the consistency levels supported by Ginger and supported by the target data service. Currently, Ginger only supports the consistency levels directly available in the target data store. Cassandra Database system supports the linear and eventual consistency levels.

`ServiceOperation<T>` represents a service operation that returns an instance of `Query<T>`. It is an extension of the `Function<Objects[], Query<T> >` interface. To obtain a reference to the implementation of the operation supplied by the `Deployed Data Service`, to which the current session is connected to, it is necessary to query the session itself.

The developer can assign consistency level to data by implementing the interfaces developed for each consistency level implemented in the Ginger system and supported by the database system in use. An example of this can be seen in listing 3.3 where `BankAccount` implements the LinearConsistency interface. To attribute a consistency level to an object, it must implement one of the consistency level interfaces provided by the Ginger API: `LinearConsistency, EventualConsistency`. Those interfaces extend the `ConsistencyLevelObject` type.

When using the `newOperation` method providing the consistency level explicitly, the consistency level of the operation will be the defined consistency level. When using the `newOperation` without explicitly defined consistency level, the consistency level of the operation will be the `ConsistencyLevelObject`'s consistency level.

A small example of a transaction with a single operation that creates a bank account follows:

```
Transaction transaction = new Transaction() {
    public void code() {
      BankAccount account = new BankAccount(1);
      newOperation(ConsistencyLevel.LINEAR,
        ↪  session.getService()::insertBankAccount, account, 10000);
```

```
1 Transaction transaction = new Transaction() {
2       public void code() {
3             Result<List<Integer>> b1 = newOperation(ConsistencyLevel.LINEAR,
              ↪  session.getService()::getAllAccountNumbers);
4       }
5 }
```

Listing 3.6: Create a new operation with no arguments and a return value

```
5     };
6 };
```

The list of objects passed in the `newOperation` method represents the parameters that the `ServiceOperation` passed will make use of. The developer can pass no objects if the operation does not need any parameters. An example of this can be seen in Listing 3.6 where the goal is to obtain a list of all account numbers of the bank system previously mentioned and it does not need parameters to obtain that information.

The return of the `newOperation` method is a `Result` object. The type of this object is the type of the expected result from the operation. If the operation is not expected to return any result then the type of the `Result` is `Void`. The `Result` contains two useful pieces of data on the operation:

- The `Integer` representative of the id of that operation;
- A `Future` object containing the response from the database, which allows the execution to not block awaiting the responses, allowing the system to interact in an asynchronous manner.

> **Java's Future object**
>
> `Future` represents the result of an asynchronous computation. The result only returns when the computation has completed [32].

3.4.3.1 Dealing with Dependencies

When dealing with dependencies between operations developers have two possible approaches depending on the type of dependencies.

Data Dependencies: Dependencies are implicitly created on two cases: when `Result` or `ConsistencyLevelObject` are passed as an argument of the method `newOperation`. When `ConsistencyLevelObject` is passed as an argument, it creates a dependency between the operation that uses it and previously created operations that also use that object.

When Result is passed as an argument, it creates a dependency between the operation that generated that Result and the operation that uses it. The Future object in Result represents the response from the database. This allows this result to be used in other operations before the operation is executed. The Result object can be passed as a parameter on newOperation method and the API will ensure that the operation will execute after the operation of the Future of Result. This is possible due to properties of the Future class. The Future class represents the result of an asynchronous computation, and when requesting the result from the Future object will ensure it is provided only when that asynchronous computation is terminated.

The processing of both Result and ConsistencyLevelObject objects passed as arguments of the operation will be further explained when we approach the processing of operation in Subsection 3.5.1

Control Dependencies: Control Dependencies must be explicitly defined by the developer. The identifier in Result is useful to determine direct dependencies between operations. Conveying control dependencies between operations is done by calling the dependentOn method, where the parameters are firstly the Result of the operation that depends on the execution of other operations, and, secondly, the Result of the operations it depends on. An example of this can be seen in listing 3.7 where the operation that will deposit 500 euros on the account number 3 is dependent on the operations that will deposit 500 on accounts number 1 and 2. This creates a dependency from operation 3 to operations 1 and 2.

3.4.4 Scope

The Ginger system also supports the creation of scopes inside transactions. Scopes allows the developer to create a scope of operations. Scopes are added to a transaction by calling the newScope method of the Transaction class and providing a Scope object. This method receives as parameters a Scope object:

```
1 public void newScope(Scope s)
```

Using the dependentOn method and providing a scope as parameter creates dependencies to other operations and scopes, similar to using the method with only operations. This way the developer ensures that those operations execute before the scope. As it can be seen in listing 3.8, line 6 to 9, the creation of a scope requires the implementation of a method named code. The implementation of this method is similar to the implementation of code in the Transaction object. The developer can use the methods available for the creation of operations inside the code method of the scope. The methods for operation creation inside the scope are the same methods provided by the Transaction object. The Scope can contain a set of operations and other operation related objects.

```
1  [...] //omitted
2  Transaction t = new Transaction() {
3      public void code() {
4          Result<Double> account1Balance=newOperation(ConsistencyLevel.EVENTUAL,
           ↪ middlewareSession.getService()::getBalance, account1);
5          Result b1=newOperation(ConsistencyLevel.EVENTUAL,
           ↪ middlewareSession.getService()::deposit, account1,
           ↪ account1balance,500.0);
6          Result<Double> account2Balance=newOperation(ConsistencyLevel.EVENTUAL,
           ↪ middlewareSession.getService()::getBalance, account2);
7          Result b2=newOperation(ConsistencyLevel.EVENTUAL,
           ↪ middlewareSession.getService()::deposit, account2,
           ↪ account2balance,500.0);
8          Result<Double> account3Balance=newOperation(ConsistencyLevel.EVENTUAL,
           ↪ middlewareSession.getService()::getBalance, account3);
9          Result b3=newOperation(ConsistencyLevel.EVENTUAL,
           ↪ middlewareSession.getService()::deposit, account3,
           ↪ account3balance,500.0);
10         dependentOn(b3,b1,b2);
11     }
12 }
```

Listing 3.7: Direct assignment of dependencies example

```
1  [...] omitted
2  Transaction transaction = new Transaction() {
3      public void code() {
4          newOperation(ConsistencyLevel.EVENTUAL,
           ↪ middlewareSession.getService()::withdraw, account1, 500.0);
5          newScope(new Scope() {
6              public void code() {
7                  newOperation(ConsistencyLevel.EVENTUAL,
                   ↪ middlewareSession.getService()::insertBankAccount,
                   ↪ account2);
8                  newOperation(ConsistencyLevel.LINEAR,
                   ↪ middlewareSession.getService()::deposit, account2, 500.0);
9              }
10         });
11     };
12 };
```

Listing 3.8: Scope creation example

3.4.5 Condition and Loop

To provide a higher level of programming the Ginger system supports loops and conditions.

3.4.5.1 Conditional

To create a new if/else condition the Transaction class provides a method called newConditional that requires as parameters, a Callable object of type Boolean, which represents the if condition part of the conditional, and one or two Scope objects depending if it is an if condition or an if/else condition:

```
1 public Result newConditional(Callable<Boolean> ifCond, Scope ifScope);
2 public Result newConditional(Callable<Boolean> ifCond, Scope ifScope, Scope
  ↪ elseScope);
```

Java's Callable object

The Callable class is a Java Util class and represents a task that returns a result. [33].

In this context the Scope class is used to represent the operations inside the if scope and, if necessary, the operations inside the else scope.

We can see an example of a conditional in listing 3.9 where the developer tries to withdraw from a bank account but only if that account has enough money for that withdrawal. Also in a conditional it is necessary to define dependencies explicitly even if the dependency is on a future object. We can see that on the example: the dependentOn method is called to establish a dependency between the conditional and the operation that returns the account.

3.4.5.2 Loop

To create a new loop, the Transaction class provides a method called newLoop that receives as parameters, a ConditionScope object and a Scope:

```
1 Result newLoop(ConditionScope loopCond, Scope loopScope)
```

The ConditionScope class represents the condition part of the loop. It is composed by a Scope and a Callable. In this case the Callable represents the condition to be checked by the loop and the Scope part of the object allows the developer to make loop verifications on database data. This means that both parts of the ConditionScope must

```
1  [...] //omitted
2  Transaction transaction = new Transaction() {
3      public void code() {
4          Result<Double> accountBalance = newOperation(ConsistencyLevel.EVENTUAL,
           ↪ middlewareSession.getService()::getBalance, account);
5          Callable<Boolean> callable = new Callable<Boolean>() {
6              public Boolean call() throws InterruptedException,
               ↪ ExecutionException {
7                  return accountBalance.get() >= 500;
8              }
9          };
10         Scope scope = new Scope() {
11             public void code() {
12                 newOperation(ConsistencyLevel.LINEAR,
                   ↪ middlewareSession.getService()::withdraw, account,
                   ↪ accountBalance, 500.0);
13             }
14         };
15         Result conditional =newConditional(callable, scope);
16         dependentOn(conditional, accountBalance);
17     };
18 };
```

Listing 3.9: Conditional Example

work together. In listing 3.10 the loop condition verifies the amount of money in a bank account. This verification is made up of a database operation, to obtain the account balance, in the Scope part and a comparison in the `Callable` part.

The Scope part of the loop works the same way as the if scope part of the conditional previously mentioned in Subsection 3.4.5.1. In the loop this scope will execute as many times as the condition is verified true.

3.4.6 Annotation Processor

To ease the burden of the programmer, Ginger also offers an annotation-based programming model that translates annotations into calls to the `front-end API`. The programmer may mark methods as transactions with Java annotations. The annotation processor was not a main objective of this thesis and, so, it is still under development at the time of this writing. The annotation processor is implemented in the scope of an `Eclipse Plugin`, by making use of the supplied Java model [34]. The use of the eclipse plugin allows for a simpler way to process ifs and loop statements, which loose their structure in `Java's byte` code. In this Section we will only approach the declaration of services and transactions.

In the context of this thesis the authors made use of both single-element annotations

```
1  [...] omitted
2  Transaction t2 = new Transaction() {
3      public void code() {
4          ConditionScope conditionScope = new ConditionScope() {
5              Result<Double> accountBalance;
6                  @Override
7                  public Boolean call() throws Exception {
8                      return accountBalance.get()<10000;
9                  }
10                 @Override
11                 public void code() {
12                     accountBalance = newOperation(ConsistencyLevel.LINEAR,
                       ↪  middlewareSession.getService()::getBalance, account);
13                 }
14         };
15         Scope scope = new Scope() {
16             public void code() {
17                 Result<Double> accountBalance =
                   ↪  newOperation(ConsistencyLevel.EVENTUAL,
                   ↪  middlewareSession.getService()::getBalance, account);
18                 newOperation(ConsistencyLevel.LINEAR,
                   ↪  middlewareSession.getService()::deposit, account,
                   ↪  accountBalance, 500.0);
19             }
20         };
21         newLoop(conditionScope, scope);
22     };
23 };
```

Listing 3.10: Loop example

Java Annotations

Java annotations are markers that associate information with a program construct. There are 3 main types of annotations:normal/marker/single-element. The normal annotation specifies a name of an annotation type and, optionally, a list of comma-separated element-value pairs, for example `@Time(hours=30,minutes=10)` or `@Square()`. The marker annotation is a shorter version of the normal annotations being represented just by @TypeName where Typename is the name of the annotation type. The single-element annotation is a shorter version of the normal annotations when it only uses a single element, as is represented by `@TypeName(ElementValue)` where Typename is the name of the annotation type and `ElementValue` is the value of the single element of the annotation [35].

Table 3.1: API Annotations

Annotation	Description
@Transaction	Identifies a Transaction to be parsed.
@Consistency(ConsistencyLevel consistencylevel)	Identifies an Object or operation with a Consistency Level passed as an argument.
@DataService(String dataServiceName)	Identifies the usage of a data service.
@Ginger	Identifies a interface to be processed as a `DataService`.

and marker annotations. The annotations allow the processor to identify the transactions, so that it can translate the developer code into code that the API can process for execution by the middleware. The annotations that the developer can use are shown in Table 3.1 with a description simply identifying their main use.

When using the Ginger annotations to create a `ConsistencyLevelObject` the developer can annotate the class with the `@Consistency(...)` annotation:

```
1 @Consistency(ConsistencyLevel.LINEAR)
2 public class BankAccount  {[...] //omitted}
```

This generates a processed class:

```
1 public class BankAccount implements LinearConsistency {[...] //omitted}
```

To use the Ginger annotations for transaction execution the interfaces for the operations must be annotated with the `@Ginger` annotation:

```
1  @Ginger
2  public interface ABankService  {
3      public  void insertBankAccount(int accountNumber);
4      BankAccount getAccount(int accountNumber) ;
5      @Consistency(ConsistencyLevel.EVENTUAL)
6      Double getBalance(BankAccount account) ;
7      void withdraw(BankAccount account,double accountBalance, float amount) ;
8      @Consistency(ConsistencyLevel.EVENTUAL)
9      void deposit(BankAccount accountB,double accountBalance, float amount);
10 }
```

This generates processed code for the interface for the operations interface:

```
1 interface GeneratedABankService  {
2     interface Ginger extends org.edgegarden.ginger.dataservice.DataService {
3         public  Query<Void> insertBankAccount(Object... opData);
4         public  Query<Void>  deleteBankAccount(Object... opData) ;
5         public  Query<Void>  updateBalance(Object... opData);
```

```
6           public  Query<Void>  depositWithAccount(Object... opData) ;
7           @Consistency(ConsistencyLevel.EVENTUAL)
8           public  Query<Void>  deposit(Object... opData) ;
9           public  Query<Void>  withdrawWithAccount(Object... opData) ;
10          public  Query<Void>  withdraw(Object... opData) ;
11          public  Query<Void>  testOperation(Object... opData) ;
12          @Consistency(ConsistencyLevel.EVENTUAL)
13          public  Query<Double> getBalance(Object... opData) ;
14          public  Query<BankAccount> getAccount(Object... opData) ;
15      }
16      public  void insertBankAccount(int accountNumber);
17      BankAccount getAccount(int accountNumber) ;
18      void withdraw(BankAccount account,double accountBalance, float amount) ;
19      @Consistency(ConsistencyLevel.EVENTUAL)
20      void deposit(BankAccount accountB,double accountBalance, float amount);
21      @Consistency(ConsistencyLevel.EVENTUAL)
22      Double getBalance(BankAccount account) ;
23  }
```

By omission the consistency level of the operations will be the highest of the `ConsistencyLevelObjects` it operates over. If it does not operate over `ConsistencyLevelObjects` the consistency level of the operations will be the highest supported by the data store. However, the consistency levels can be directly indicated, as in lines 5-6 and 8-9 of the previous example, by using the `@Consistency(...)` annotation.

To use the Ginger annotations processor the user must identify the transaction code to be parsed with the `@Transaction` annotation:

```
1   public class Bank {
2       @DataService("Cassandra Ginger Bank")
3       ABankService service;
4       @Transaction
5       void transfer(BankAccount account1, BankAccount account2, float amount) {
6           Double account1Balance = service.getBalance(account1);
7           service.withdraw(account1,account1Balance, amount);
8           Double account2Balance = service.getBalance(account2);
9           service.deposit(account2,account2Balance, amount);
10      }
11  }
```

The code of the transaction contains the operations. To create an operation the developer must create a data service (as seen in lines 2 and 3), by instantiating a data service interface that was annotated with `@Ginger`, as previously mentioned. The creation of a Ginger operation is done by calling one of the method of the data service.

The code generated by the processor when processing the example above is as follows:

```
1  class ProcessedBank {
2      Session<GeneratedABankService.Ginger> service =
3              new LocalSession<GeneratedABankService.Ginger>(
               ↪  GeneratedABankService.Ginger.class, "Cassandra Ginger Bank");
4      void transfer(BankAccount account1, BankAccount account2, float amount) {
5          new org.edgegarden.ginger.middleware.objects.Transaction() {
6              @Override
7              public void code() {
8                  Result<Double> account1Balance =
                   ↪  newOperation(ConsistencyLevel.EVENTUAL,
                   ↪  service.getService()::getBalance, account1);
9                  newOperation(service.getService()::withdraw,account1,
                   ↪  account1Balance, amount);
10                 Result<Double> account2Balance =
                   ↪  newOperation(ConsistencyLevel.EVENTUAL,
                   ↪  service.getService()::getBalance, account2);
11                 newOperation(ConsistencyLevel.EVENTUAL,
                   ↪  service.getService()::deposit, account2, account2Balance,
                   ↪  amount);
12             }
13         };
14     }
15 }
```

The generated code is Ginger API code for creation of a transaction. This code will then be processed by the API and furthermore executed by the middleware.

3.5 Processing Transactions

Transactions are pre-processed at the client before being sent to the middleware layer for execution. This processing stage is triggered when asking a session to run a transaction (line 18 of Listing 3.11).

The transaction processing algorithm begins by adding the transaction to the set of *transactions under execution*. Afterwards the algorithm calls the method code. A transaction scope contains a set of ordered lists, one per consistency level supported by the session. The lists registers the operations and other execution objects. For a Cassandra session, the transaction will contain a set of two lists, one for each consistency level.

3.5.1 Processing Operations

When the code (`method code`) of a transaction is run, every call to `newOperation` creates two objects: one of type `Operation`, to represent the operation, and a second, of type `Result`, to represent the response to the execution of that operation. The `Result` is

```
1  Session<BankService> middleware = new
   ↪ LocalSession<BankService>(BankService.class, "Cassandra Ginger Bank");
2  BankAccount account1= new BankAccount(1);
3  BankAccount account2= new BankAccount(2);
4  BankAccount account3= new BankAccount(3);
5  Transaction t = new Transaction() {
6      public void code() {
7      // withdraw
8      Result<Double> account1Balance = newOperation(ConsistencyLevel.EVENTUAL,
       ↪ middleware.getService()::getBalance, account1);
9      newOperation(ConsistencyLevel.LINEAR,  middleware.getService()::withdraw,
       ↪ account1, account1Balance,500.0);
10     //transfer
11     Result<Double> account2Balance =
12     newOperation(ConsistencyLevel.EVENTUAL, middleware.getService()::getBalance,
       ↪ account2);
13     newOperation(ConsistencyLevel.LINEAR, middleware.getService()::withdraw,
       ↪ account2, account2Balance, 500.0);
14     Result<Double> account3Balance = newOperation(ConsistencyLevel.EVENTUAL,
       ↪ middleware.getService()::getBalance, account3);
15     newOperation(ConsistencyLevel.LINEAR, middleware.getService()::deposit,
       ↪ account3, account3Balance, 500.0);
16     };
17 };
18 session.run(t);
19 session.close();
```

Listing 3.11: Full transaction example

returned by the execution of newOperation, as it was mentioned in Subsection 3.4.3. This returned object contains a Future object, which will await for the response of the execution of the operation's query.

Once created, the Operation object is added to one of the consistency level lists of the Transaction. The storage of the operation is dependent on the consistency level of the operation, either determined explicitly or implicitly by the ConsistencyLevelObject, as mentioned in Subsection 3.4.3.

Upon creation of an operation, the algorithm analyzes the parameters of the operation and checks for dependencies to other operations.

The algorithm checks this in two cases:

- If the operation contains one or more Result objects as a parameter;
- If the operation contains one or more ConsistencyLevelObject objects as a parameter.

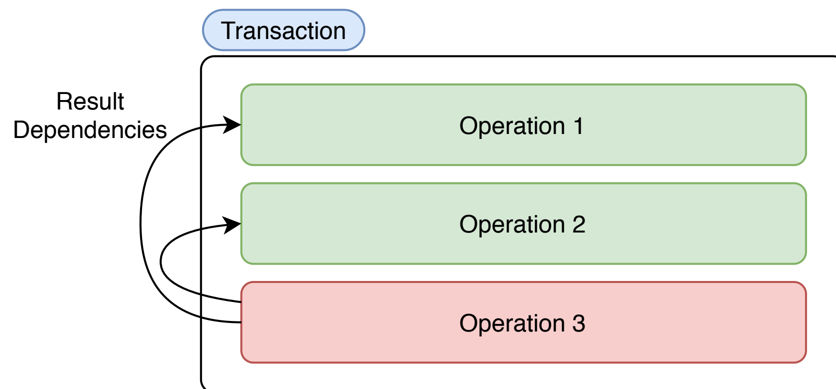

Figure 3.2: Transaction graphical example of `Result` dependencies (The operations painted green are of a weaker consistency level then the one painted red.).

If the operation received one or more Result objects as arguments, these objects are analyzed to obtain the operation from which it was generated, so that it can establish a dependency between both operations. If that dependency exists, the algorithm assures that the operation generating that `Result` executes before the operation that depends on it, by raising the consistency level of the operation that generates the response, to the consistency level of the operation that depends on that response. This does not happen if the operation generating the response already has a consistency level equal or higher to the consistency level of the operation depending on that response.

Looking at the example of figure 3.2, which is an example of the execution of a transaction that has 3 operations. The operations painted green are of a weaker consistency level then the one painted red. In the example the operations 1 and 2 are green operations (weaker consistency level) and operation 3 is a red operation (stronger consistency level). Operation number 3 contains as a parameter a `Result` object from both operation 1 and 2.

The processing of that transaction as exemplified in figure 3.3 happens as follows:

- Firstly, operation 1 is added to the weaker consistency level list of operations;
- Secondly, operation 2 is added to the weaker consistency level list of operations;
- Finally, upon the creation of operation 3, it is verified that it contains `Result` objects from both previous operations. Based on that, both operations consistency levels are raised to the stronger consistency level, and both are moved to the stronger consistency level list. Following that operation 3 is added to that list too. The operations after the change still maintain the original order of creation.

If the operation contains one or more `ConsistencyLevelObject` objects, as a parameter, the algorithm will verify already stored operation objects to check if other operations

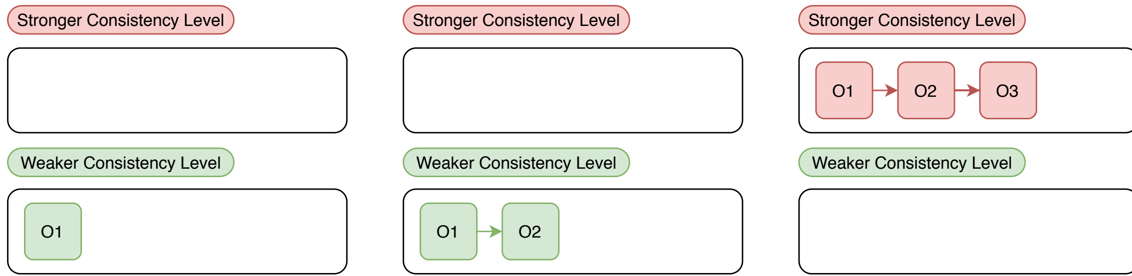

Figure 3.3: Diagram representative of the processing of a transaction with dependencies resulting from usage of results from operations.(The operations painted green are of a weaker consistency level then the one painted red.)

make accesses to that same `ConsistencyLevelObject`. If this checks true, the algorithm verifies the consistency level of both operations. If the operation that occurs first has a weaker consistency level than the operation that occurs after, the first operation consistency level is raised to the consistency level of the operation that occurs after. Therefore, this ensures that the first operation executes before the last operation. This prevents data hazards such as wrong Read after write (RAW), Write after read (WAR) and Write after write (WAW), but it is also an overly safe approach as it also acts if both operations are reads. We plan to make future work on this area and try to achieve an approach that provides safety only for the necessary dependencies.

Looking at the example of figure 3.4, which is an example of the execution of a transaction that has 4 operations. As in the previous example, operations identified with green are of weaker consistency level than operations identified with red. In the example the operations 1, 2 and 4 are green operations (weaker consistency level) and operation 3 is a red operation (stronger consistency level). Operation number 3 contains as a parameter a `ConsistencyLevelObject` object with strong consistency level, that operation number 1 also has as a parameter.

The processing of that transaction as exemplified in figure 3.5 happens as follows:

- Firstly, operation 1 is added to the weaker consistency level operations list;
- Secondly, operation 2 is added to the weaker consistency level operations list;
- Then, upon the creation of operation 3, it is verified that this operation contains `ConsistencyLevelObject` object of strong consistency level that operation 1 also uses. This creates a dependency between them both, resulting in the necessity to raise operation 1 consistency level and moving it to the stronger consistency level list of operations. The operations after the change still maintain the original order of creation.
- Finally, operation 4 is added to the weaker consistency level list of operations.

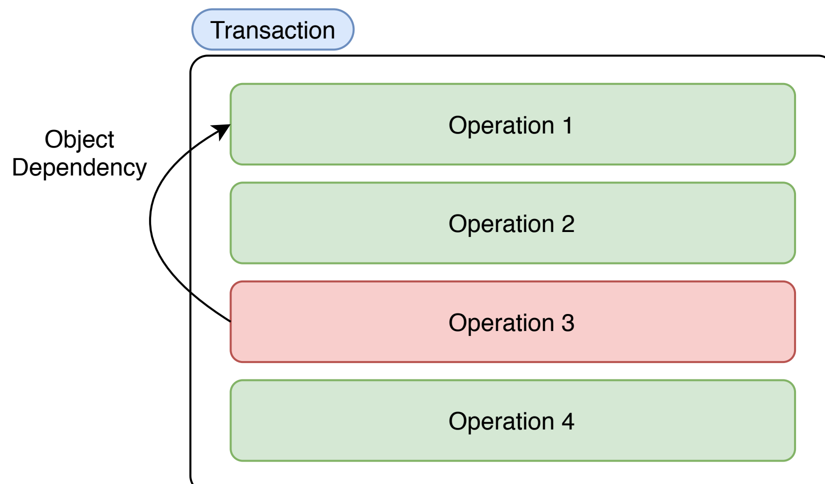

Figure 3.4: Transaction graphical example of ConsistencyLevelObject dependencies (The operations painted green are of a weaker consistency level then the one painted red.).

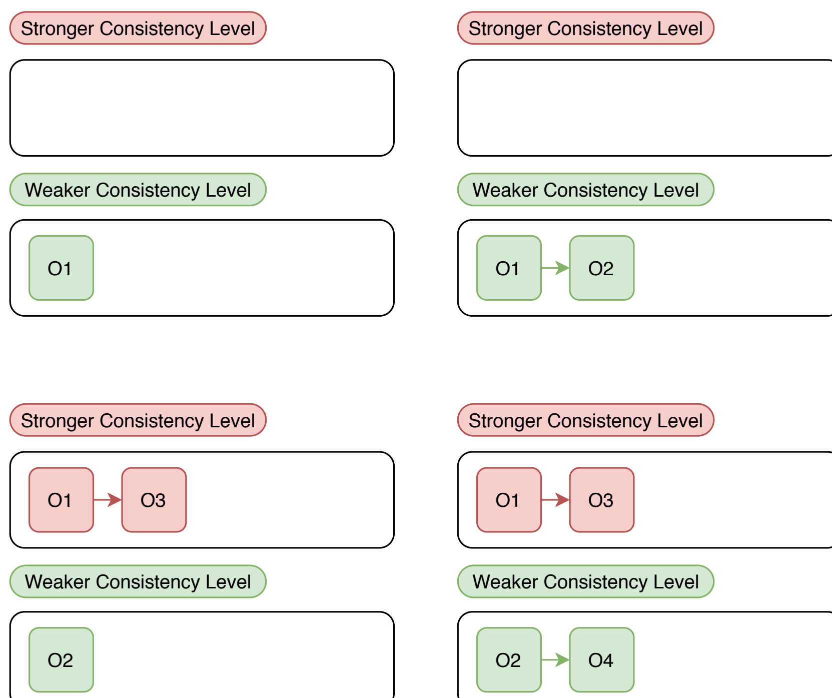

Figure 3.5: Diagram representative of the processing of a transaction with dependencies resulting from usage of ConsistencyLevelObject objects (The operations painted green are of a weaker consistency level then the one painted red.).

For both cases of dependencies previously analyzed, when there is a dependency between 2 operations, where operation 1 occurs before operation 2 and operation 2 is dependent on operation 1: if operation 1 consistency levels is lower than operation 2 consistency level, operation 1 consistency level is raised but if operation 1 consistency level is higher or equal to operation 2 consistency level nothing is altered. The way the dependencies are dealt between operations is not an ideal approach, but it was the safest approach for the state of implementation. Raising the consistency level of an operation to ensure it occurs before another may raise problems, specially with time concerns. If an operation was supposed to execute in a weaker consistency level but after processing is set to execute at a higher consistency level may result in longer execution times. The authors suggest another approach, for research and testing in future work, that is, instead of raising the consistency level of the operation, saving the notion of the dependency and during execution ensuring that the dependency is not broken by waiting to execute an operation till the operation it depends on finishes execution. This may be a problem as the stronger consistency level operations are set to execute prior to weaker consistency level operations and if a stronger consistency level operation is to occur after a weaker consistency level operation that it depends on may cause a deadlock. This case needs to be studied during further implementation of the Ginger middleware system.

3.5.2 Processing Scopes

The algorithm when processing a scope, and this is, when the algorithm is processing the code method of the transaction and the `newScope` method is called, processes the new object representative of the scope, as previously referenced in Subsection 3.4.4.

The algorithm when processing a `Scope` processes it as if it was a transaction for the most part. It starts by processing the code (`code` method). The algorithm processes this method the same way it processes the code method of `Transaction`. When processing a `Scope`, the system divides operations of the scope by consistency level and adds a copy of the scope to each list of the Transactions consistency level lists. Each instance of the `Scope` only contains the operations with consistency level equal to the list the `Scope` is in.

Looking at the example of figure 3.6, which is an example of the processing of a scope that has 3 operations. As in previous examples, operations identified with green are weaker consistency level operations and operations identified with red are stronger consistency level operations. In the example, the operations 1 and 2 are green operations (weaker consistency level) and operation 3 is a red operation (stronger consistency level). All the operations are inside the scope.

The processing of the scope of the previous example happens as follows:

- First, the scope is processed and the operations are separated into the consistency level lists. Operations one and two are put in the weaker consistency level list and operation three is put in the stronger consistency level list;

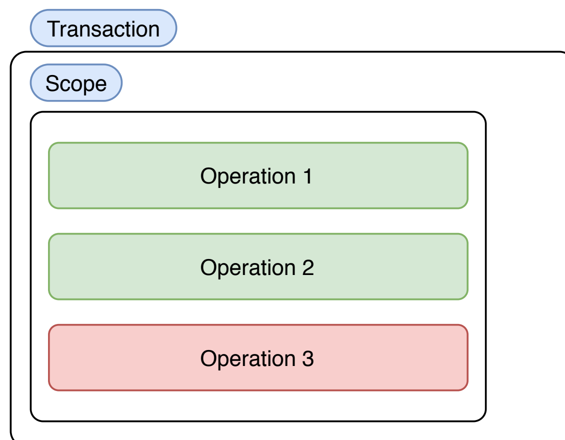

Figure 3.6: Diagram representative of a Scope with three operations.

- After the scope is processed the algorithm divides the scope in as many scopes as there are consistency level lists with operations. In this case there are two consistency level lists. The algorithm then creates a scope containing the operations of the weaker consistency level list and another scope with the stronger consistency level list;
- Finally, the scopes that were created by the algorithm are added to the corresponding consistency level lists in the `Transaction`.

We can see the final state of this processing in figure 3.7.

3.5.3 Processing Conditions

The algorithm when processing a condition, meaning, when the algorithm is processing the code method of the transaction and the `newConditional` method is called, creates a new object representative of a condition, an object of the type `Conditional`. As previously mentioned in Subsection 3.4.5, the `Conditional` is made of a `Callable` object of the type `Boolean` and one or two Scope objects depending if it is an `if` condition or an `if/else` condition. Upon creation of the `Conditional` object, it is attributed a consistency level. This consistency level is the higher consistency level of the consistency levels of both scopes. The dependencies between the `Conditional` and operations external to it must be explicitly defined as mentioned in Subsection 3.4.5. Based on this, upon creation, the `Conditional` is added to one of the transaction lists, based on its consistency level. The algorithm does not make any verification of dependencies on `Conditional` objects. This approach is not ideal as it ignores potential data hazards, as there may be operations modifying data in different stages. This is due to the fact that there may be operations, both inside and outside the `Conditional`, modifying data at different consistency levels, which may cause undesired conditions such as endless waiting and operations executing in an undesired order. The solution to this problem is to explicitly defined those dependencies

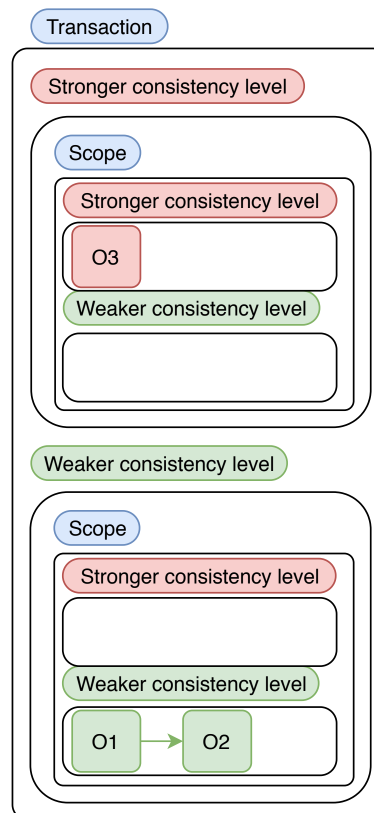

Figure 3.7: Diagram representative of the processing of a scope.

as mentioned in Subsection 3.4.5. The authors suggest another approach, for research and testing in future work, where the dependencies do not have to be explicitly set. In this approach the `Conditional` would be divided, so that, for each consistency level it executes only the operations of that consistency level. This means creating replicas of the `Conditional` object for each consistency level, containing only the operations of that consistency level. Also, in this approach, the operations of the `Conditional` should be checked for dependencies to operations outside the `Conditional`.

3.5.4 Processing Loops

When processing a loop the algorithm processes the call of `newLoop` method. It creates a new object that represents a loop, an object of type `Loop`. The loop is composed of a `ConditionScope` and a `Scope`. As with the `Conditional` the `Loop` is attributed a consistency level upon creation. The consistency level of the loop is the higher consistency level from both the `ConditionScope` and the `Scope`. Also with loop the dependencies must be explicitly defined between the loop and operations that are external to the loop. Upon creation, the `Loop` is added to the list of operations, based on its consistency level, and no verification of dependencies is made by the algorithm. The problem with this approach is similar to the problem with the approach to the `Conditional`. It poses a

potential threat to the consistency of the data, as multiple operations both inside and outside the `Loop` may access the same `ConsistencyLevelObject` in different consistency levels, which may lead to accesses that should happen after certain operations to happen before, resulting in inconsistencies of order in the database. The solution suggested by the authors to this problem is the same solution previously mentioned in Subsection 3.5.3 for the `Conditional`. In future work is recommended the research and testing of a solution where there dependencies do not need to be explicitly defined. The suggested solution is to create multiple versions of the loop, one for each consistency level the operations of the scope use. This would allow the loop to execute in different consistency levels without forcing the highest consistency level for all operation. Even though this approach seems to be the best, it poses serious problems when it comes to the verification of the condition of the loop and the number of times the loop executes. The coordination of this approach requires the system to keep state versions for the loop in the different consistency levels. The approach shall also have into account the necessity to verify the dependencies between the operations inside the loop and the operations outside the loop, treating the operations inside the loop as if they were normal operations outside the loop.

3.6 Middleware

The goal of the middleware as it was explained in Section 3.2 is to be deployed in a distributed manner, allowing for implementation in different scenarios and services. By compartmentalizing the Ginger system we allow the developer's system to be more efficient and allow for multiple front-end services to communicate through the same middleware, to the back-end services. This is a more efficient and logic approach, saving developer resources as the developer can implement one middleware to work with all of its front-end services.

There can also be multiple instances of the middleware deployed working together, with each instance communicating at least with one data store. The middleware instances ensure, through coordination, that the operations execute under the defined consistency levels. The goal is for the middleware instances to support more consistency levels than the consistency levels supported by the data store, and, when a consistency level required by the developer is not supported by the database, the middleware instances coordinate to provide that consistency level. In the execution of this thesis this was not a priority. For the future of Ginger this functionality will be developed.

The distributed approach also allows for further implementation of APIs for other coding languages other than the already implemented Java approach. Even though this is the ideal approach as of now this is not implemented yet. As mentioned in Subsection 3.4.1 the current state of the middleware is working together with the front-end API as one but there is already the start of an implementation for an approach using REST, which would allow the middleware to be deployed in a distributed manner.

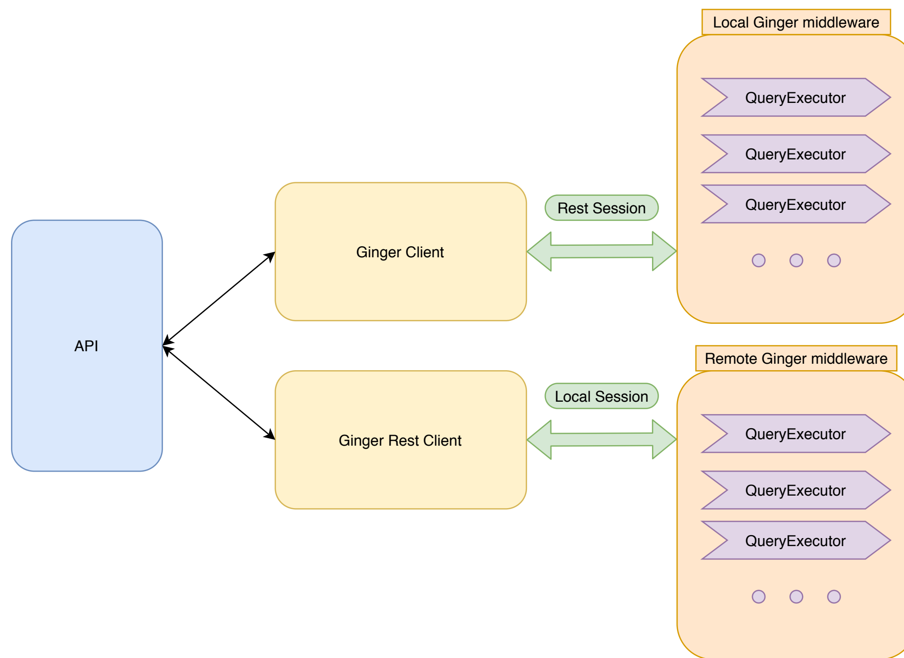

Figure 3.8: Diagram representative of the sequence of execution of a transaction.

After the processing of a transaction, a process that was described in Section 3.5, the API calls an execution method. The execution of a transaction by the API occurs by calling a client that is coupled with the API. This client then, based on the session created at the start by the developer, as mentioned in Subsection 3.4.1, communicates with the middleware. There are two implemented clients. The goal of these clients is to communicate with the middleware. One of the clients is for usage with a local version of the middleware, this is, if the developer is using the middleware together with the API. The other implemented client is for usage with a distributed version of the middleware. This version of the client uses REST communication to cooperate with the middleware. The client then communicates the processed transactions to the middleware through the session the middleware established with the API. Inside the middleware, there are algorithms to process each transaction, these algorithms are called `Query Executor`. Their main goal is to execute the operations of a transaction and commit them to the database. In Figure 3.8 we can see a representative diagram of this implementation of the sequence of execution of a transaction.

The middleware receives the transaction split in sets of operations. A `Query Executor` then executes the operation in a decreasing order of strength of consistency level, that is, for example, with the two consistency levels currently implemented in the Ginger system, the middleware will then start by executing the operations of linear consistency level and after concluding those executions, it will then execute the operations of eventual consistency level. The `Query Executor` when executing operations for consistency levels it allows for execution of operations in an asynchronous manner. The implemented eventual consistency level is an example of this, as the `Query Executor` executes the

operation in parallel using an Executor. An Executor is a Java Util class with the functionality of executing threads. Then, for the eventual consistency level the Query Executor processes the operations asynchronously and commits them to the database only with the requirement that one of the replicas responds. For stronger consistency levels and in the implemented case, the linear consistency level, the Query Executor executes the operations linearly, executing only one operation at a time and committing them with the requirement that all the replicas must respond.

3.7 Back-end Communication

For communication with the back-end and execution of the operations, the middleware uses a connector class to connect to the database system. This is currently only implemented for the Cassandra database system, but it is implemented in a extensible manner so that in future implementations it is possible to be implemented for other database systems. There is a CassandraConnector class, which extends a Connector class. This Connector class allows for future implementation of connectors to other database systems.

The communication to the database system starts when the developer asks the middleware for a specific session. When asking for a new session the developer provides the Data Service class that will be used and a key unique to that session. The middleware, upon receiving this information, verifies if there is any connection matching the requests, as it contains a map of all the services. The middleware maps a Data Service to an IP address and a port. This way, when a developer request a session, it verifies if there is any connection between the middleware and the database system under the conditions stipulated by the developer. If so, the middleware establishes a connection to the database system, creating a connector, and establishes a session allowing for communication through the connection. The middleware makes use of this session to commit the transactions to the database. Upon committing an operation, the middleware receives the response from the database. This response will then be processed according to the function provided in the Query object by the developer, in the service, as previously mentioned in Section 3.3. This translated result is then further communicated to the API by the middleware.

After the processing of the transactions that the developer wants to commit to the database, it is recommended that the developer closes the session. This can be obtained by calling the close method after the call for execution of a transaction as we can see in listing 3.11. The close method is a method from the Session object created in the API. The closing of a session can also be forced by calling the forceClose method, closing the session even if a transaction is being committed. The example in listing 3.11 is an example of the full code for committing a single transaction using the Ginger API.

Chapter 4

Evaluation

In this Chapter we showcase the goal of the middleware that needs to be evaluated, in Section 4.1, following, in Section 4.2, with a description of how to evaluate the set goal. Furthermore, we will explore the tests that were realized until the end of development of this thesis, in Section 4.3.

4.1 Objective

The Ginger middleware system innovates by providing a transaction coordination system that allows for the attribution of consistency levels in both the operations and the data objects. This difference is what needs to be evaluated in comparison with other systems. Also it is ideal to test the system against different versions of himself to test the difference between approaches to how consistency is defined in transactions. To achieve this we made minor changes to the system so that it can be tested using both consistency level definition over the data and the operations, against a version of the system using only consistency level defined by the data and a version of the system using consistency level defined by the transactions. This comparison would have the goal of showing if the Ginger system brings any advantages over other transaction coordination systems and languages, for example the MixT Language for Mixing Consistency in Geodistributed Transactions [3]. The system must be tested with the following properties in consideration:

- Can the system correctly split transactions?
- Can the system maintain a correct ordering between transaction that does not impact consistency negatively?
- Latency;

- Does the system impact negatively the systems and applications that are using it?
- Do the systems and applications using Ginger show benefits from using it?.

4.2 Methodology

To test the system, Ginger was deployed in a node of a cluster, in communication with a back-end service, which was comprised of three nodes running CassandraDB system in replication.

To evaluate Ginger, we searched for benchmarks that simulate the kind of systems we are targeting. Rubis [22], TPC-W [23] and Yahoo! Cloud Serving Benchmark (YCSB) were obvious candidates, but as also mentioned in MixT [3], these do not yet embed the notion of transaction with multiple consistency levels. To adapt one, or more, of such benchmarks to serve our needs would be a very interesting addition, but, unfortunately, this could not be done in the time span of the thesis.

Accordingly, we chose to implement a simple benchmark that mimics the basic transactions of a bank system: transfers, deposits, withdrawals and balance verification. Our benchmark randomly executes these transactions based on a chosen probability for each type. It executes a total of 5 runs for 5 minutes each. Finally, upon ending, it provides a set of results containing:

- Number of operations executed;
- Number of eventual operations executed;
- Number of linear operations executed;
- Number of transactions executed;
- Number of operations executed over `BankAccount`;
- Number of log operations executed;
- Average transaction execution time.

To ensure the currently developed system is well implemented, we produced a set of tests that will evaluate the first 2 properties, previously mentioned in Section 4.1: "Can the system correctly split transactions" and "Can the system maintain a correct ordering between transaction that does not impact consistency negatively". We also ran our benchmark, comparing three version of our system: one with consistency defined over data, one with consistency defined over data and operations and one with consistency defined over transactions.

The Ginger system was tested in two ways. Firstly, it was tested for functionality, this is, it was tested to confirm if all the implemented functionalities are working as supposed, if the operations are executing in the right order and if the operations are executing with

the predicted consistency levels. Secondly, we ran our benchmark, comparing three version of our system with consistency defined over data, with consistency defined over data and operations and with consistency defined over transactions. Both these testing scenarios are analysed in Section 4.3, Subsections 4.3.1 and 4.3.2 respectively.

4.3 Conducted tests

4.3.1 Functionality tests

The Ginger system was tested for correctness by executing a set of tests from which some are worth analyzing:

- Propagation of data dependencies;
- Propagation of control dependencies;
- Execution of scopes;
- Propagation of dependencies explicitly;
- Execution of conditions;
- Execution of loops;

Future objects test: The purpose of this first test is twofold: checks if the system correctly propagates dependencies between operations where one of the operation uses the result from another operation as a parameter and checks if the consistency level is raised in case of a dependency. In the case of an operation of stronger consistency using a response value from an operation of weaker consistency, the test has the goal of verifying if the weaker consistency operation is raised in level and happens before the other operation, as mentioned in 3.5.1. The transaction used for testing is the one partially exemplified in listing 4.1. In this transaction an account is created and added to the database, followed by a balance verification and a withdrawal.

From the transaction we can see that the `withdraw` operation on line 9 uses the balance obtained from the `getBalance` operation on line 8, which means that the consistency level of the `getBalance` operation must be raised from eventual to linear and this operation must execute before the `withdraw` operation. The execution of the previously mentioned transaction produces the output seen in listing 4.2.

From the output we can conclude that in fact, the `getBalance` operation represented in line 2 executes before the `withdraw` operation represented in line 3 and that the `getBalance` operation executed with linear consistency level. The test executed as supposed demonstrating the predicted and that the system works as supposed when propagating this dependency.

```java
[...] // omitted
Transaction t = new Transaction() {
    public void code() {
        BankAccount account = new BankAccount(1);
        //insert bank account
        newOperation(ConsistencyLevel.LINEAR,
         ↪ session.getService()::insertBankAccount, account, 10000);
        //withdrawal
        BackendResponse<Double> accountBalance =
         ↪ newOperation(ConsistencyLevel.EVENTUAL,
         ↪ session.getService()::getBalance, account);
        newOperation(ConsistencyLevel.LINEAR, session.getService()::withdraw,
         ↪ account, accountBalance, 500.0);
    };
};
session.run(t);
[...] // omitted
```

Listing 4.1: Future test.

```
Executing operation: INSERT INTO bankAccounts (accountNumber,balance) VALUES
 ↪ (1,10000.0); with consistency: Consistency Linear
Executing operation: SELECT balance FROM bankAccounts WHERE accountNumber=1;
 ↪ with consistency: Consistency Linear
Executing operation: UPDATE bankAccounts SET balance=9500.0 WHERE
 ↪ accountNumber=1; with consistency: Consistency Linear

Total number of operations executed: 3
Total number of linear operations executed: 3
Total number of eventual operations executed: 0
```

Listing 4.2: Future test output.

```
1  [...] // omitted
2  Transaction t = new Transaction() {
3      public void code() {
4          BankAccount account = new BankAccount(1);
5          //insert bank account
6          newOperation(session.getService()::insertBankAccount, account, 10000);
7          newOperation(ConsistencyLevel.EVENTUAL, session.getService()::deposit,
           ↪  account, 500.0);
8          newOperation(session.getService()::withdraw, account, 500.0);
9      };
10 };
11 session.run(t);
12 [...] // omitted
```

Listing 4.3: ConsistencyLevelObject test.

```
1  Executing operation: INSERT INTO bankAccounts (accountNumber,balance) VALUES
   ↪  (1,10000.0); with consistency: Consistency Linear
2  Executing operation:  with consistency: Consistency Linear
3  Executing operation:  with consistency: Consistency Linear
4  Total number of operations executed:3
5  Total number of linear operations executed:3
6  Total number of eventual operations executed:0
```

Listing 4.4: ConsistencyLevelObject test output.

ConsistencyLevelObject test: Tests if the system correctly propagates dependencies between operations that use the same `ConsistencyLevelObject` as a parameter. If an operation of weaker consistency level that uses a `ConsistencyLevelObject` of strong consistency as a parameter and happens before an operation of stronger consistency level that uses the same `ConsistencyLevelObject` as a parameter, the test has the goal to verify that the system raises the consistency level of the weaker consistency level operation so that both operation execute in the implemented order. This is the intended processing under the conditions explained in Subsection 3.5.1. The transaction used for testing is the one partially exemplified in listing 4.3. In this transaction an account is created and inserted to the database. After, the test executes two operations a deposit and a withdrawal from the bank account;

Analyzing the transaction, we can see that the operation at line 7 requires eventual consistency and receives `account` as argument; a `ConsistencyLevelObject`. Next, the operation at line 8 requires linear consistency and also receives `account` as an argument. The execution of that transaction produces the output seen in listing 4.4.

The output of the transaction is as predicted, the operation of eventual consistency

```
1  [...] // omitted
2  Transaction t = new Transaction() {
3      public void code() {
4          newScope(new Scope() {
5              @Override
6              public void code() {
7                  BankAccount account1 = new BankAccount(1);
8                  BankAccount account2 = new BankAccount(2);
9                  BankAccount account3 = new BankAccount(3);
10                 // insert bank accounts
11                 newOperation(ConsistencyLevel.LINEAR,
                   ↪  session.getService()::insertBankAccount, account1, 10000);
12                 newOperation(ConsistencyLevel.LINEAR,
                   ↪  session.getService()::insertBankAccount, account2, 10000);
13                 newOperation(ConsistencyLevel.LINEAR,
                   ↪  session.getService()::insertBankAccount, account3, 10000);
14                 // balance checks
15                 newOperation(ConsistencyLevel.EVENTUAL,
                   ↪  session.getService()::getBalance, account1);
16                 newOperation(ConsistencyLevel.EVENTUAL,
                   ↪  session.getService()::getBalance, account2);
17                 newOperation(ConsistencyLevel.EVENTUAL,
                   ↪  session.getService()::getBalance, account3);
18             }
19         });
20     };
21 };
22 session.run(t);
23 [...] // omitted
```

Listing 4.5: Scope test.

level was raised to the linear consistency level as we can see, in lines 2 and 3, that both of the test operations were executed at the linear consistency level. The test executed as pretended demonstrating the predicted execution and that propagation of dependencies between operations sharing data is working as intended.

Scope test: Tests the system for correct execution of a scope. In a scope the operations are split just as in a transaction and on execution they execute the same way they would if there was no scope. This is the intended processing, as explained in Subsection 3.5.2. The transaction used for testing this is the one partially exemplified in listing 4.5. In this transaction a scope is created. This scope creates 3 accounts, adds them to the database and finishes by verifying the balance of the created accounts.

The execution of that transaction produces the output seen in listing 4.6.

```
1 Executing operation: INSERT INTO bankAccounts (accountNumber,balance) VALUES
  ↪ (1,10000.0); with consistency: Consistency Linear
2 Executing operation: INSERT INTO bankAccounts (accountNumber,balance) VALUES
  ↪ (2,10000.0); with consistency: Consistency Linear
3 Executing operation: INSERT INTO bankAccounts (accountNumber,balance) VALUES
  ↪ (3,10000.0); with consistency: Consistency Linear
4 Executing operation: SELECT balance FROM bankAccounts WHERE accountNumber=1;
  ↪ with consistency: Consistency Eventual
5 Executing operation: SELECT balance FROM bankAccounts WHERE accountNumber=2;
  ↪ with consistency: Consistency Eventual
6 Executing operation: SELECT balance FROM bankAccounts WHERE accountNumber=3;
  ↪ with consistency: Consistency Eventual
7 Total number of operations executed:6
8 Total number of linear operations executed:3
9 Total number of eventual operations executed:3
```

Listing 4.6: Scope test output.

The output produced the expected result executing firstly the linear operations for creating the accounts, represented in lines 1,2 and 3 of the output, followed by the accounts balance verification, represented in lines 4, 5 and 6 of the output.

Directly defined dependencies test: When testing the dependentOn method, which allows the API to directly define dependencies between operations, the test must verify if the operations defined in the method occur in the supposed order. This is, as mentioned in 3.4.3, the dependentOn method creates a dependency where the first operation passed as an argument of the method is dependent on the operations that come after. If the operation that depends on the others has a stronger consistency level than any of those operations, the operations with lower consistency level have their consistency levels raised so that they occur before the operation that depends on them. The transaction used for testing this situation is the one partially exemplified in listing 4.7. In this transaction two accounts are created and then there are two operations to insert them to the database, followed by a dependency so that the second account is added after the first account.

This transaction has two database operation to add two accounts to the database: one executing in the eventual consistency level and another in the linear consistency level. After the creation of the two operations, there is a dependency created, in line 8, between the operations in lines 6 and 7. The execution of that transaction produces the output seen in listing 4.8.

From executing the transaction in listing 4.7, the output in listing 4.8 was produced and it is the intended output has the dependency that was created resulted in the first insert to the database in line 1 of the output to happen in linear consistency and before the second insert that is in line 2 of the output.

```
1  [...] // omitted
2  Transaction t = new Transaction() {
3      public void code() {
4          BankAccount account = new BankAccount(1);
5          BankAccount account2 = new BankAccount(2);
6          BackendResponse op1=newOperation(ConsistencyLevel.EVENTUAL,
           ↪ session.getService()::insertBankAccount, account, 10000);
7          BackendResponse op2=newOperation(ConsistencyLevel.LINEAR,
           ↪ session.getService()::insertBankAccount, account2, 10000);
8          dependentOn(op2, op1);
9      };
10 };
11 session.run(t);
12 [...] // omitted
```

Listing 4.7: Directly defined dependencies test.

```
1  Executing operation: INSERT INTO bankAccounts (accountNumber, balance) VALUES
   ↪ (1, 10000.0); with consistency: Consistency Linear
2  Executing operation: INSERT INTO bankAccounts (accountNumber, balance) VALUES
   ↪ (2, 10000.0); with consistency: Consistency Linear
3  Total number of operations executed:2
4  Total number of linear operations executed:2
5  Total number of eventual operations executed:0
```

Listing 4.8: Directly defined dependencies test output.

Condition test: When testing a condition, there is the necessity to verify if the algorithm analyzes the condition correctly and executes the corresponding piece of code. This is, as mentioned in 3.4.5, the creation of a condition requires the implementation of a `Callable` to be the condition part and at least the implementation of one Scope to be the corresponding code of the condition. The transaction used for testing this condition is the one partially exemplified in listing 4.9. In this transaction an account is created and added to the database. Following, the balance of the added account is verified. If the balance is greater then 10000 euros then an amount of 500 is withdrawn.

From this transaction, we can see, in line 5, that the account added has a balance of 10080 euros, which is higher then the 10000 of the condition, making the condition true and resulting in the execution of the scope, that starts in line 12. The execution of that transaction produces the output seen in listing 4.10.

When analyzing the output we can verify that the algorithm tested true for the condition since the withdrawal operation occur. The withdrawal operation is represented by the query in line 4. The condition executed as supposed, demonstrating that the condition implemented in the Ginger system is working as it was described in Subsection 3.5.3

```
[...] // omitted
Transaction t = new Transaction() {
    public void code() {
        BankAccount account = new BankAccount(777);
        BackendResponse b1 =newOperation(ConsistencyLevel.EVENTUAL,
         ↪ session.getService()::insertBankAccount, account, 10080);
        BackendResponse<Double> b2 = newOperation(ConsistencyLevel.EVENTUAL, ,
         ↪ session.getService()::getBalance, account);
        Callable<Boolean> c = new Callable<Boolean>() {
            public Boolean call() throws InterruptedException,
             ↪ ExecutionException {
                return b2.get() > 10000;
            }
        };
        Scope s1 = new Scope() {
            public void code() {
                BackendResponse<Double> balance =
                 ↪ newOperation(ConsistencyLevel.EVENTUAL,
                 ↪ session.getService()::getBalance, account);
                newOperation(ConsistencyLevel.LINEAR,
                 ↪ session.getService()::withdraw, account, balance, 500.0);
            }
        };
        BackendResponse c1 =newConditional(c, s1);
        dependentOn(c1,b1,b2);
    };
};
session.run(t);
[...] // omitted
```

Listing 4.9: Condition test.

```
Executing operation: INSERT INTO bankAccounts (accountNumber,balance) VALUES
 ↪ (777,10080.0); with consistency: Consistency Linear
Executing operation: SELECT balance FROM bankAccounts WHERE accountNumber=777;
 ↪ with consistency: Consistency Linear
Executing operation: SELECT balance FROM bankAccounts WHERE accountNumber=777;
 ↪ with consistency: Consistency Linear
Executing operation: UPDATE bankAccounts SET balance=9580.0 WHERE
 ↪ accountNumber=777; with consistency: Consistency Linear
Total number of operations executed:4
Total number of linear operations executed:4
Total number of eventual operations executed:0
```

Listing 4.10: Condition test output.

```
1  [...] // omitted
2  Transaction t = new Transaction() {
3      public void code() {
4          BankAccount account = new BankAccount(777);
5          BackendResponse b1 = newOperation(ConsistencyLevel.LINEAR,
           ↪  session.getService()::insertBankAccount, account, 9998);
6          ConditionScope cs = new ConditionScope() {
7              BackendResponse<Double> accountbalance;
8              @Override
9              public Boolean call() throws Exception {
10                 return accountbalance.get()<10000;
11             }
12             @Override
13             public void code() {
14                 accountbalance = newOperation(ConsistencyLevel.LINEAR,
                   ↪  session.getService()::getBalance, account);
15             }
16         };
17         Scope s1 = new Scope() {
18             public void code() {
19                 BackendResponse<Double>
                   ↪  accountBalance=newOperation(ConsistencyLevel.LINEAR,
                   ↪  session.getService()::getBalance, account);
20                 newOperation(ConsistencyLevel.LINEAR,
                   ↪  session.getService()::deposit,account, accountBalance, 1.0);
21             }
22         };
23         BackendResponse c1 = newLoop(cs, s1);
24     };
25 };
26 session.run(t);
27 [...] // omitted
```

Listing 4.11: Loop test.

Loop test: When testing a loop, the test must verify if the loop executes correctly and the number of times it is supposed to execute. This is, as mentioned in 3.4.5, the creation of a loop requires the implementation of a ConditionScope to be the condition part and the implementation of one Scope to be executed in the loop. The transaction used for testing the loop is the one partially exemplified in listing 4.11. In this transaction an account is created and the account is added to the database. Further, the balance of the added account is verified. If the balance of the account is less then 10000 euros then an amount of one euro is deposit into that account.

From this transaction we can see, in line 3, that the account added as balance of 9998

```
1  Executing operation: INSERT INTO bankAccounts (accountNumber,balance) VALUES
   ↪ (777,9998.0); with consistency: Consistency Linear
2  Executing operation: SELECT balance FROM bankAccounts WHERE accountNumber=777;
   ↪ with consistency: Consistency Linear
3  Executing operation: SELECT balance FROM bankAccounts WHERE accountNumber=777;
   ↪ with consistency: Consistency Linear
4  Executing operation: UPDATE bankAccounts SET balance=9999.0 WHERE
   ↪ accountNumber=777; with consistency: Consistency Linear
5  Executing operation: SELECT balance FROM bankAccounts WHERE accountNumber=777;
   ↪ with consistency: Consistency Linear
6  Executing operation: SELECT balance FROM bankAccounts WHERE accountNumber=777;
   ↪ with consistency: Consistency Linear
7  Executing operation: UPDATE bankAccounts SET balance=10000.0 WHERE
   ↪ accountNumber=777; with consistency: Consistency Linear
8  Executing operation: SELECT balance FROM bankAccounts WHERE accountNumber=777;
   ↪ with consistency: Consistency Linear
9  Total number of operations executed:8
10 Total number of linear operations executed:8
11 Total number of eventual operations executed:0
```

Listing 4.12: Loop test output.

euros, which is less than the 10000 euros of the condition of the loop. This results in the condition of the loop executing three times and the code of the loop executing twice. The execution of that transaction produces the output seen in listing 4.12.

From the output of the transaction we can conclude that the loop executed the expected 2 times, with the condition part being represented by the operations in lines 2, 5 and 8, and the code of the loop being represented in lines 3,4,6 and 7. The loop executed as supposed. demonstrating that the loop in Ginger is working as it was described in Subsection 3.5.4

4.3.2 Consistency tests

Testing the advantages of the Ginger approach to consistency, of assigning consistency over data and operations, it is both complicated at this stage, due to the system not being completely finished, but is also difficult to compare since the advantages it brings are not only of execution but also of implementation. Independently of these constraints, a large test was realized.

Test Description The test consists of comparing three different scenarios of execution. The scenarios used for comparison vary on the way the consistency is define but all of them are implementations of the Ginger system. The tests were realizes over the bank system previously mentioned and used in Chapter 3. The bank system used for testing

represents a simple database system with a set of bank accounts and a set of logs. In the Ginger example, the bank accounts are represented by a class BankAccount that was previously shown in listing 3.3 The test uses the BankCassandraService example that is shown in listing 3.4. In that example there are omitted operations, one of which will be useful for the comprehension of the tests realized. The operation omitted was:

```
public  Query<Void> addLog(Object... opData) {
        Log b= (Log) opData[0];
        Insert insert = QueryBuilder.insertInto( logTable ).value(
         ↪  "logId" ,b.hashCode()).value( "logMessage",
         ↪  b.getLogMessage() );
        return new Query<Void>(insert);
    }
```

In this operation an object Log, which is a ConsistencyLevelObject of the type EventualConsistency is committed to the database. This is used to register a transaction. The Log object is a simple object containing an integer identifier and a string with a message representing the transaction.

The first scenario of testing was the Ginger system with consistency defined over operations and data. This scenario is the normal implementation of Ginger and the expected utilization of the system. The second scenario of testing was the Ginger system with consistency defined only over data. In this scenario the consistency of the operations is defined by the consistency level of the objects they interfere with, which in this case was the bank account, for which the consistency level is linear, and the log, for which the consistency level is eventual. The last scenario of testing was the Ginger system with consistency defined over transaction. In this scenario the consistency of the operations is defined by the transactions, which means that if one of the operations inside the transaction requires a higher level of consistency all the operations will execute at that higher level of consistency.

For each of the previously mentioned scenarios, a set of tests was realized. The set of tests are equal between the scenarios. This set of tests consists in testing the scenario for a set of different type of transactions with different probabilities for each type of transactions. For each test there were created ten bank accounts.

The types of transactions executed were:

- Bank transfers between two random accounts;
- Bank deposits to a random accounts;
- Bank withdrawals from a random account;
- Bank balance checks of a random account.

For each scenario the tests were executed for three sets of probabilities:

Table 4.1: Node specification

Node	CPU	Total Cores/ Threads	Memory	Network	Operative System	Main Disk
Node 1	Intel Xeon X3450	4/8	8 GiB DDR3 1333 MHz	2 x 1 Gbps	Ubuntu 19	230 GB HDD
Node 2	Intel Xeon X3450	4/8	8 GiB DDR3 1333 MHz	2 x 1 Gbps	Ubuntu 19	230 GB HDD
Node 3	Intel Xeon X3450	4/8	8 GiB DDR3 1333 MHz	2 x 1 Gbps	Ubuntu 19	230 GB HDD
Node 4	2 x Intel Xeon E5-2620 v2	12/24	64 GiB DDR3 1600 MHz	2 x 1 Gbps	Ubuntu 19	100 GB SSD

- 40% transfers, 20% deposits, 20% withdrawals, 20% balance checks;
- 20% transfers, 30% deposits, 30% withdrawals, 20% balance checks;
- 20% transfers, 20% deposits, 20% withdrawals, 40% balance checks.

Each of these sets of tests was executed five times for the duration of five minutes. The tests realized generate a set of result which represent: the number of operations of each type of consistency level that were executed, the number of transactions executed, the number of operation executed over `BankAccount`,the number of log executed and the average execution time of a transaction. The tests were executed in a cluster of nodes, using four nodes: Three nodes for database replication with equal specification and 1 for execution of the middleware. The three nodes used for database replication ran the `Cassandra database` and shared a key space with a replication factor of three, which means the data of that key space is saved in all three nodes. The specification of the nodes are depicted in table 4.1. Nodes 1 to 3 were used for the database system and node 4 was used for Ginger system.

The transaction for creating an account is a simple transaction. As seen in the example of listing 4.13, the transaction starts by creating a new account with an account identifying integer number and a `Double` value representative of the initial value of the balance of the account.

The transaction for executing a transfer between two accounts is, as seen in listing 4.14, made by obtaining the value of the balance of the first account, followed by the withdrawal of the money to be transferred which in the case of the test was 500 euros. After, it obtains the balance value of the second account and terminates by depositing the transfer value in the second account.

```
1  [...] // omitted
2  Transaction t = new Transaction() {
3      public void code() {
4          BankAccount account = new BankAccount(ACCOUNTNUMBER);
5          newOperation(ConsistencyLevel.LINEAR,
           ↪  middleware.getService()::insertBankAccount, account, 1000000);
6          Log log=new Log("Account created with account number
           ↪  "+account.getNumber()+" and balance "+account.getBalance());
7          newOperation(ConsistencyLevel.EVENTUAL,middleware.getService()::addLog,log);
8      };
9  };
10 middleware.run(t);
11 [...] // omitted
```

Listing 4.13: Creating bank account example.

```
1  [...] // omitted
2  Transaction t = new Transaction() {
3      public void code() {
4          BackendResponse<Double>
           ↪  account1Balance=newOperation(ConsistencyLevel.EVENTUAL,
           ↪  session.getService()::getBalance, account1);
5          newOperation(ConsistencyLevel.LINEAR, session.getService()::withdraw,
           ↪  account1, account1Balance, 500.0);
6          BackendResponse<Double>
           ↪  account2Balance=newOperation(ConsistencyLevel.EVENTUAL,
           ↪  session.getService()::getBalance, account2);
7          newOperation(ConsistencyLevel.EVENTUAL, session.getService()::deposit,
           ↪  account2, account2Balance, 500.0);
8          Log log=new Log("Transaction of " + 500.0 + " euros between account
           ↪  number " + account1Number + " and account number " +
           ↪  account2Number);
9          newOperation(ConsistencyLevel.EVENTUAL ,session.getService()::addLog,
           ↪  log);
10     };
11 };
12 session.run(t);
13 [...] // omitted
```

Listing 4.14: Transfer transaction example.

```
1  [...] // omitted
2  Transaction t = new Transaction() {
3      public void code() {
4          BackendResponse<Double>
           ↪ accountBalance=newOperation(ConsistencyLevel.EVENTUAL,
           ↪ session.getService()::getBalance, account);
5          newOperation(ConsistencyLevel.EVENTUAL, session.getService()::deposit,
           ↪ account, accountBalance, 500.0);
6          Log log=new Log("Deposit of " + 500.0 + " euros to account number " +
           ↪ account.getNumber());
7          newOperation(ConsistencyLevel.EVENTUAL, session.getService()::addLog,
           ↪ log);
8      };
9  };
10 session.run(t);
11 [...] // omitted
```

Listing 4.15: Deposit transaction example.

The transaction for depositing an amount of money into an account is made of a balance check of the account, followed by a deposit operation for a certain amount of money which in the case of this test was the value of 500 euros. This transaction is represented in the example of listing 4.15.

The transaction for withdrawing money from an account balance is similar to the deposit transaction. It is composed of an operation to verify the account balance and an operation to withdraw the amount of money from the balance of the account. This transaction can be seen, exemplified, in listing 4.16

The transaction for realizing a balance verification in an account, as seen in listing 4.17, is simply made of an operation where the user provides the account object to obtain the balance of that account.

Test Results Each test from the previously described tests generates 7 results: the total number of operations executed, the number of eventual operations executed, the number of linear operations executed, the number of operations executed over a bank account object, the number of operations executed over a log, the average time of execution of a transaction and the total number of transactions executed. This means that for each scenario a total of 105 results were generated. As previously mentioned, each test of each set of percentages was run five times during five minutes to obtain valid results. The results of those five executions were averaged to be compared between scenarios. All the tests, for all three scenarios, were developed so that the number of eventual operations was as high as possible.

```
1  [...] // omitted
2  Transaction t = new Transaction() {
3      public void code() {
4          BackendResponse<Double>
           ↪  accountBalance=newOperation(ConsistencyLevel.EVENTUAL,
           ↪  session.getService()::getBalance, account);
5          newOperation(ConsistencyLevel.LINEAR, session.getService()::withdraw,
           ↪  account, accountBalance, 500.0);
6          Log log=new Log("Withdrawal of " + 500.0 + " euros from account number "
           ↪  + account.getNumber());
7          newOperation(ConsistencyLevel.EVENTUAL, session.getService()::addLog,
           ↪  log);
8      };
9  };
10 session.run(t);
11 [...] // omitted
```

Listing 4.16: Withdraw transaction example.

```
1  [...] // omitted
2  Transaction t = new Transaction() {
3      public void code() {
4          BackendResponse<Double>
           ↪  accountBalance=newOperation(ConsistencyLevel.EVENTUAL,
           ↪  session.getService()::getBalance, account);
5          Log log=new Log("Balance check on account number " +
           ↪  account.getNumber());
6          newOperation(ConsistencyLevel.EVENTUAL, session.getService()::addLog,
           ↪  log);
7      };
8  };
9  session.run(t);
10 [...] // omitted
```

Listing 4.17: Balance verification transaction example.

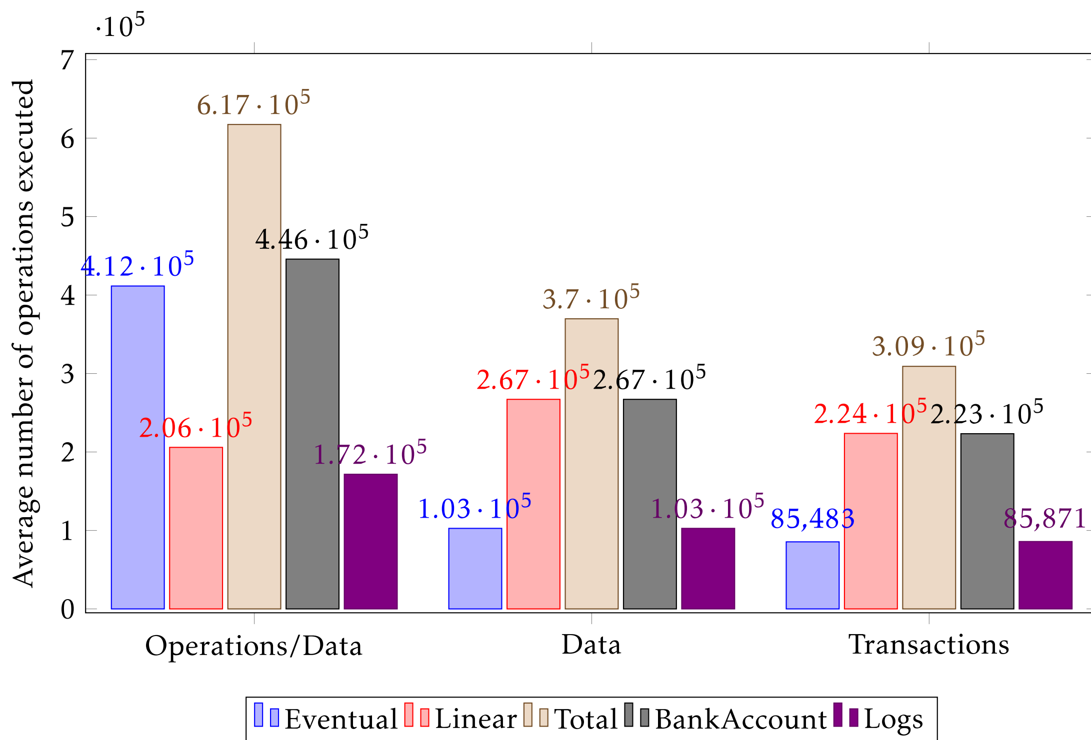

Figure 4.1: Average number of operations for each consistency level, for the execution of a test with a higher percentage of transfers.

For the test with a higher value of transfers (40% transfers, 20% deposits, 20% withdrawals, 20% balance checks) the values obtained for the number of operations executed are depicted in the graph of Figure 4.1. For this test the average transaction execution time for the execution of the 5 iterations of the test was:

- 1.725 milliseconds for the scenario with consistency defined over operations and data
- 2.900 milliseconds for the scenario with consistency defined over Data
- 3.466 milliseconds for the scenario with consistency defined over Transactions

The average total number of transactions executed for the execution of the 5 iterations of the test was:

- 171614 transactions executed for the scenario with consistency defined over operations and data
- 102654 transactions executed for the scenario with consistency defined over Data
- 85871 transactions executed for the scenario with consistency defined over Transactions

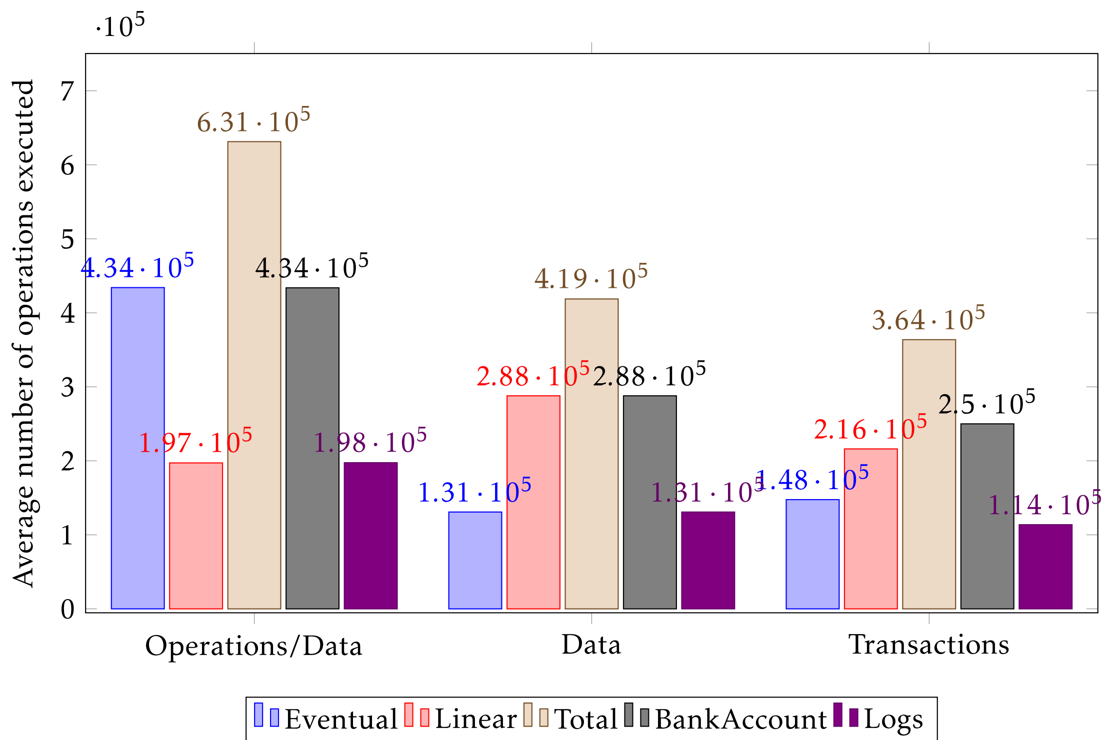

Figure 4.2: Average number of operations for each consistency level, for the execution of a test with a higher percentage of deposits and withdrawals.

For the test with a higher value of deposits and withdrawals (20% transfers, 30% deposits, 30% withdrawals, 20% balance checks) the values obtained for the number of operations executed are depicted in the graph of Figure 4.2. For this test the average transaction execution time for the execution of the 5 iterations of the test was:

- 1.497 milliseconds for the scenario with consistency defined over operations and data
- 2.270 milliseconds for the scenario with consistency defined over Data
- 2.614 milliseconds for the scenario with consistency defined over Transactions

The average total number of transactions executed for the execution of the 5 iterations of the test was:

- 197509 transactions executed for the scenario with consistency defined over operations and data
- 130854 transactions executed for the scenario with consistency defined over Data
- 113730 transactions executed for the scenario with consistency defined over Transactions

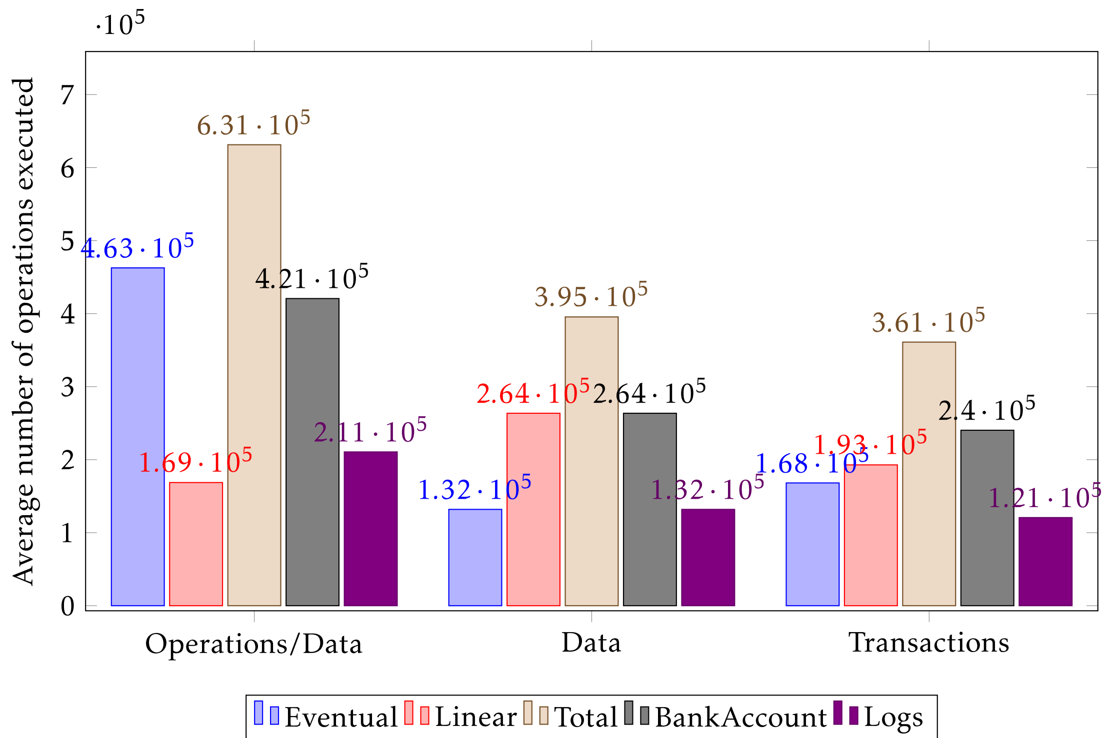

Figure 4.3: Average number of operations for each consistency level, for the execution of a test with a higher percentage of balance checks.

For the test with a higher value of balance checks (20% transfers, 20% deposits, 20% withdrawals, 40% balance checks) the values obtained for the number of operations executed are depicted in the graph of Figure 4.3. For this test the average transaction execution time for the execution of the 5 iterations of the test was:

- 1.4021 milliseconds for the scenario with consistency defined over operations and data
- 2.2490 milliseconds for the scenario with consistency defined over Data
- 2.4614 milliseconds for the scenario with consistency defined over Transactions

The average total number of transactions executed for the execution of the 5 iterations of the test was:

- 210644 transactions executed for the scenario with consistency defined over operations and data
- 131918 transactions executed for the scenario with consistency defined over Data
- 120700 transactions executed for the scenario with consistency defined over Transactions

When analyzing the results we can see a clear difference between the three approaches. On all the three tests the results between scenarios are very consistent. From the obtained results we can conclude that, overall, the Ginger approach of using consistency defined over operations and data shows the best results. This is due to the fact that using that approach allowed for higher number of eventual operations. A higher number of eventual operation executed allowed for a faster transaction execution and the execution of a higer number of transactions, as we can see, from all three test realized, that the average transaction execution time for the Ginger approach was clearly lower than both the other approaches. Comparing the approach using consistency defined over data and operations with the approach using consistency over data, we can see a much lower number of eventual transactions on the second approach. This is due to the fact that `BankAccount` object is a linear consistency object and most of the operations are over the `BankAccount` object. The eventual operations executed in this approach are resultant from the log operations, since the Log object is a `ConsistencyLevelObject` of `EventualConsistency` type. The approach using only consistency defined over transactions also presents similar results to the previously discussed approach. This is a consequence of this approach using only one consistency level for a whole transactions. For example in a transfer of money where Ginger would realize two linear operations and three eventual operations, the approach using consistency over transactions with execute all 5 operations with linear consistency. When comparing this approach to Ginger approach there is still a clear advantage of the Ginger approach, as this approach executes a higher number of transaction and executes transactions at faster speeds.

This is a minor demonstration that the Ginger system allows for a more fluid and faster execution of transactions and does not limit the consistency level of operations as much as a system with consistency over data or consistency over transactions. Even though this result is promising, we acknowledge that this is a limited test, since it only realizes tests on one system and limited circumstances. There is the necessity for further testing on this matter, and this test only serves as an incentive for further development of the system under the conditions currently implemented. Further testing may reveal different results and the development will have the necessity to adapt to those conditions.

Chapter 5

Conclusion

This chapter has two sections. The first section presents a conclusion on the results analyzed on chapter 4 and the process of implementation of the Ginger system. The second section provides some insight on the necessary and planned work for the future of Ginger along with some improvements suggestions.

5.1 Conclusion

In this dissertation, we present our approach on a development of a middleware system for coordination of consistency levels of database system transactions. Along with the development of the middleware we also developed an API. The implemented system uses consistency levels defined both over the operations of the transaction and the data of the transactions. Based on our research we can conclude that Ginger is the only system that takes this approach when it comes to defining consistency in transactions. Ginger provides a set of tools that allows users to create transactions and commit them with the desired consistency levels. The Ginger system was inspired in a variety of systems that we analyzed in chapter 2, but its development was mainly inspired in the `MixT` [3] system. This inspiration resulted in Ginger taking a similar, but yet different approach, as Ginger splits transactions by consistency levels, the same way that `MixT` does, but Ginger also provides consistency levels defined over operations. Ginger is also a system with an `API` and middleware system, whereas `MixT` is a language for mixed consistency in transactions. Ginger provides different ways to create transactions by either using the API methods or the provided API annotations.

Ginger system development is not yet finished as this thesis focuses on a proof of concept and the basis for further development of this system. The development described in chapter 3, as substantiated by the tests in chapter 4, proved that the usage of consistency

defined over operations and data in a transaction has advantages and can result in a more fluid execution of transactions in comparison with systems that use only consistency defined over data or consistency defined over transactions.

In the end, by analyzing the evaluation results, we can conclude that the proposed development of the structure for a middleware system for coordination with different levels of consistency proved to be advantageous in some regards, comparing to other systems that differ from Ginger on the way the consistency is defined. This thesis serves as an incentive of further investigation into this approach to consistency coordination and for further development of the Ginger system.

5.2 Future Work

Ginger is a working middleware system for coordination of consistency levels in transactions but to some extent it is only a prototype as it is still missing some features we find necessary to be deployed. Also, the system, due to being the first development where the goal was to provide a working system, would benefit from both some improvements in execution and bigger variety of utilization possibilities. Thus, we propose a set of aspects to be considered for future work, split in two main groups: system development and system improvements.

5.2.1 System development

In this subsection we focus on the main development areas that need to be worked on in future implementation of the Ginger system.

Ginger Rest In its current state, the Ginger middleware has to be deployed together with the Ginger API, as mentioned in subsection 3.4.1. Thus, for future development, we suggest concluding the already started implementation of both the REST client and REST session. Also the development of a means of communication in the middleware is necessary. With this developments the goal is to establish a connection between the middleware and the API allowing both to communicate via REST. This approach allows the middleware to be deployed separately from the API, allowing the user to establish one middleware for multiple APIs and multiple resources.

Rollback As of current development there is no way of rolling back transactions. This is a clear problem as leaving a transaction not fully committed may result in major damage to the data in the database and result in large inconsistencies in the database. For further development, we recommend further investigation and development on a way to rollback transactions, and we also suggest this to be one of the main points of focus in further development.

System testing We briefly discussed, in section 4.3, the importance of realizing more tests and tests that verify both the correctness of the system and the effectiveness. For testing we deployed the Database system and the middleware in a local cluster but for future testing we pretended to deploy the Database systems and the middleware in a cloud system (possibly Microsoft Azure [36]), in a disperse manner, deploying them in multiple regions of the world. The current stage Ginger is on did not allow for such testing but we highly believe that this should also be one of the main points of focus for further development of the system as this may reveal potential major issues and also provide solutions to smaller problems. We recommend, that, as soon as there is a version of the middleware where the middleware is deployed detached from the API, tests are realized with the middleware deployed in a server and the API deployed in a different server. We also suggest trying to adapt a existing benchmark for this tests and if not possible create one from scratch as this testing is crucial for evaluation of the system and the development of the system.

Annotations implementation As previously mentioned, in section 3.4.6, the implementation of the annotations is not finished. We suggest for future work implementing what is missing in the annotations and altering the processing of those annotations.

5.2.2 System improvements

In this subsection we focus on the main areas the system can be improved on but that are not crucial for the working state of the Ginger system.

Processing conditions As previously discussed in subsection 3.5.3, the processing of dependencies in conditions is only done by the user explicitly defining those dependencies. That approach is not ideal and we suggest an approach similar to the one used in transactions. This is, creating multiple instances of the condition, one for each consistency level the operations of the conditional use. Those operations shall also be verified for dependencies to the operations outside the condition and treated as if they are normal operations of a transaction.

Processing loops As previously discussed in subsection 3.5.4, the processing of dependencies in loops is done by explicitly defining them on the moment of creation of the transaction. This approach, as the approach for processing conditions, is not ideal. The author suggests a solution similar to the one described in the previous paragraph. The suggested solution is to create multiple versions of the loop for each consistency level of the operations of the loop scope and analyzing for dependencies to operations outside the loop.

Multiple db The goal of the Ginger system is to support multiple database systems. As of current state, the system only supports the `Cassandra database` system. For the

future we expect the system to be further developed in this area, providing the users a way to use whichever database they prefer and the one that works best with their systems.

Multiple consistency levels Coupled with the previous suggestion for future work, for supporting more database systems, comes the support for multiple consistency levels. On the beginning of this thesis we set the goal of supporting 3 consistency levels, but during development it proved to be a unreal challenge as there were other priorities in the system and the time for development was scarce. So, for future development, we suggest the addition of a bigger variety of consistency levels to work in conjuncture with the database systems.

Dynamic consistency We briefly talked about dynamic consistency in subsection 2.4.2. The goal of the system is to support this way of treating consistency, but is far from being one of Ginger's main focus. But we still believe this is an area worth exploring as the system efficiency would benefit majorly from this execution. We suggest then that dynamic consistency is investigated and developed in a future development of the system. We recommend implementing this only after the other future work suggestions previously mentioned are implemented and working properly, as this is not as important or as crucial as any of the previous suggestions.

References

[1] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems - concepts and designs (3. ed.)* International computer science series. Addison-Wesley-Longman, 2002. ISBN: 978-0-201-61918-8.

[2] E. A. Brewer. "Towards robust distributed systems (abstract)". In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA*. Ed. by G. Neiger. ACM, 2000, p. 7. DOI: 10.1145/343477.343502. URL: https://doi.org/10.1145/343477.343502.

[3] M. Milano and A. C. Myers. "MixT: a language for mixing consistency in geodistributed transactions". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. Ed. by J. S. Foster and D. Grossman. ACM, 2018, pp. 226–241. DOI: 10.1145/3192366.3192375. URL: https://doi.org/10.1145/3192366.3192375.

[4] B. Holt et al. "Disciplined Inconsistency with Consistency Types". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*. Ed. by M. K. Aguilera, B. Cooper, and Y. Diao. ACM, 2016, pp. 279–293. DOI: 10.1145/2987550.2987559. URL: https://doi.org/10.1145/2987550.2987559.

[5] D. B. Terry et al. "Consistency-based service level agreements for cloud storage". In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. Ed. by M. Kaminsky and M. Dahlin. ACM, 2013, pp. 309–324. DOI: 10.1145/2517349.2522731. URL: https://doi.org/10.1145/2517349.2522731.

[6] P. Bailis et al. "Highly Available Transactions: Virtues and Limitations". In: *PVLDB* 7.3 (2013), pp. 181–192. DOI: 10.14778/2732232.2732237. URL: http://www.vldb.org/pvldb/vol7/p181-bailis.pdf.

[7] C. J. Date. *SQL and Relational Theory - How to Write Accurate SQL Code, Second Edition*. Theory in practice. O'Reilly, 2012. ISBN: 978-1-449-31640-2. URL: http://www.oreilly.de/catalog/9781449316402/index.html.

[8] M. Curtiss. *Why you should pick strong consistency, whenever possible*. 2018. URL: https://cloud.google.com/blog/products/gcp/why-you-should-pick-strong-consistency-whenever-possible.

[9] K. C. Sivaramakrishnan, G. Kaki, and S. Jagannathan. "Declarative programming over eventually consistent data stores". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. Ed. by D. Grove and S. Blackburn. ACM, 2015, pp. 413–424. DOI: 10.1145/2737924.2737981. URL: https://doi.org/10.1145/2737924.2737981.

[10] A. Gotsman et al. "'Cause I'm strong enough: reasoning about consistency choices in distributed systems". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by R. Bodık and R. Majumdar. ACM, 2016, pp. 371–384. DOI: 10.1145/2837614.2837625. URL: https://doi.org/10.1145/2837614.2837625.

[11] D. Bermbach and J. Kuhlenkamp. "Consistency in Distributed Storage Systems - An Overview of Models, Metrics and Measurement Approaches". In: *Networked Systems - First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers*. Ed. by V. Gramoli and R. Guerraoui. Vol. 7853. Lecture Notes in Computer Science. Springer, 2013, pp. 175–189. DOI: 10.1007/978-3-642-40148-0_13. URL: https://doi.org/10.1007/978-3-642-40148-0%5C_13.

[12] K. Petersen et al. "Bayou: replicated database services for world-wide applications". In: *Proceedings of the 7th ACM SIGOPS European Workshop: Systems Support for Worldwide Applications, 1996, Connemara, Ireland, September 9-11, 1996*. Ed. by A. Herbert and A. S. Tanenbaum. ACM, 1996, pp. 275–280. DOI: 10.1145/504450.504497. URL: https://doi.org/10.1145/504450.504497.

[13] S. Burckhardt. "Principles of Eventual Consistency". In: *Foundations and Trends in Programming Languages* 1.1-2 (2014), pp. 1–150. DOI: 10.1561/2500000011. URL: https://doi.org/10.1561/2500000011.

[14] V. Balegas et al. "Putting consistency back into eventual consistency". In: *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*. Ed. by L. Réveillère, T. Harris, and M. Herlihy. ACM, 2015, 6:1–6:16. DOI: 10.1145/2741948.2741972. URL: https://doi.org/10.1145/2741948.2741972.

[15] C. Li et al. "Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary". In: *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. Ed. by C. Thekkath and A. Vahdat. USENIX Association, 2012, pp. 265–278. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

[16] T. Kraska et al. "Consistency Rationing in the Cloud: Pay only when it matters". In: *PVLDB* 2.1 (2009), pp. 253–264. DOI: 10.14778/1687627.1687657. URL: http://www.vldb.org/pvldb/vol2/vldb09-759.pdf.

[17] C. Li et al. "Automating the Choice of Consistency Levels in Replicated Systems". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by G. Gibson and N. Zeldovich. USENIX Association, 2014, pp. 281–292. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/li%5C_cheng%5C_2.

[18] C. Li, N. M. Preguiça, and R. Rodrigues. "Fine-grained consistency for geo-replicated systems". In: *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. Ed. by H. S. Gunawi and B. Reed. USENIX Association, 2018, pp. 359–372. URL: https://www.usenix.org/conference/atc18/presentation/li-cheng.

[19] S. Roy et al. "The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by T. K. Sellis, S. B. Davidson, and Z. G. Ives. ACM, 2015, pp. 1311–1326. DOI: 10.1145/2723372.2723720. URL: https://doi.org/10.1145/2723372.2723720.

[20] Y. Sovran et al. "Transactional storage for geo-replicated systems". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. Ed. by T. Wobber and P. Druschel. ACM, 2011, pp. 385–400. DOI: 10.1145/2043556.2043592. URL: https://doi.org/10.1145/2043556.2043592.

[21] E. Hewitt. *Cassandra - The Definitive Guide: Distributed Data at Web Scale*. Springer, 2011. ISBN: 978-1-449-39041-9. URL: http://www.oreilly.de/catalog/9781449390419/index.html.

[22] *RUBiS*. URL: http://kcsrk.info/Quelea/rubis-test.html.

[23] D. A. Menascé. "TPC-W: A Benchmark for E-Commerce". In: *IEEE Internet Computing* 6.3 (2002), pp. 83–87. DOI: 10.1109/MIC.2002.1003136. URL: https://doi.org/10.1109/MIC.2002.1003136.

[24] W. Lloyd et al. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*. Ed. by T. Wobber and P. Druschel. ACM, 2011, pp. 401–416. DOI: 10.1145/2043556.2043593. URL: https://doi.org/10.1145/2043556.2043593.

[25] D. D. Akkoorath et al. "Cure: Strong Semantics Meets High Availability and Low Latency". In: *36th IEEE International Conference on Distributed Computing Systems, ICDCS 2016, Nara, Japan, June 27-30, 2016*. IEEE Computer Society, 2016, pp. 405–414. DOI: 10.1109/ICDCS.2016.98. URL: https://doi.org/10.1109/ICDCS.2016.98.

[26] J. Du et al. “GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks”. In: *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 3-5, 2014*. Ed. by E. Lazowska et al. ACM, 2014, 4:1–4:13. DOI: 10.1145/2670979.2670983. URL: https://doi.org/10.1145/2670979.2670983.

[27] M. Bravo, L. E. T. Rodrigues, and P. V. Roy. “Saturn: a Distributed Metadata Service for Causal Consistency”. In: *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*. Ed. by G. Alonso, R. Bianchini, and M. Vukolic. ACM, 2017, pp. 111–126. DOI: 10.1145/3064176.3064210. URL: https://doi.org/10.1145/3064176.3064210.

[28] S. Burckhardt et al. “Eventually Consistent Transactions”. In: *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. Ed. by H. Seidl. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 67–86. DOI: 10.1007/978-3-642-28869-2_4. URL: https://doi.org/10.1007/978-3-642-28869-2%5C_4.

[29] P. Bailis and A. Ghodsi. “Eventual consistency today: limitations, extensions, and beyond”. In: *Commun. ACM* 56.5 (2013), pp. 55–63. DOI: 10.1145/2447976.2447992. URL: https://doi.org/10.1145/2447976.2447992.

[30] J. Du, S. Elnikety, and W. Zwaenepoel. “Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks”. In: *IEEE 32nd Symposium on Reliable Distributed Systems, SRDS 2013, Braga, Portugal, 1-3 October 2013*. IEEE Computer Society, 2013, pp. 173–184. DOI: 10.1109/SRDS.2013.26. URL: https://doi.org/10.1109/SRDS.2013.26.

[31] H. Attiya, E. Hillel, and A. Milani. “Inherent Limitations on Disjoint-Access Parallel Implementations of Transactional Memory”. In: *Theory Comput. Syst.* 49.4 (2011), pp. 698–719. DOI: 10.1007/s00224-010-9304-5. URL: https://doi.org/10.1007/s00224-010-9304-5.

[32] I. Oracle America. *Future (Java Platform SE 7)*. 2015. URL: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html.

[33] I. Oracle America. *Callable (Java Platform SE 7)*. 2015. URL: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Callable.html.

[34] L. Vogel, S. Scholz, and F. Pfaff. *Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model*. 2020. URL: https://www.vogella.com/tutorials/EclipseJDT/article.html.

[35] J. Gosling et al. *The Java® Language Specification,Java SE 15 Edition*. 2020. URL: https://docs.oracle.com/javase/specs/jls/se15/html/jls-9.html#jls-9.7.

[36] *Microsoft Azure*. URL: https://azure.microsoft.com/.